Étude de la sécurité intrinsèque des langages fonctionnels (LaFoSec)

Titre	**Modèles d'exécution de OCaml**
Identifiant	**Livrable L3.1.2**
Version	**5.0**
Date	**2011-10-17**
Pages	**100**
Approbation	**Christèle Faure, SafeRiver**
	Date: Signature:

Table des révisions

Version	Date	Description et changements	Parties modifiées
1.0	2011-02-14	Version initiale correspondant à la première fourniture du livrable L3.1.1.	Tout le document
2.0	2011-04-14	Version présentée à la revue de la version 1.0 du 16 mars 2011	Tout le document
3.0	2011-09-16	Version correspondant à la première fourniture du livrable L3.1.2.	Tout le document
4.0	2011-09-28	Version incluant les remarques de l'ANSSI.	Tout le document
5.0	2011-10-17	Version prenant en compte les remarques de l'ANSSI.	Tout le document

Résumé

Le langage OCaml appartient à la famille des langages fonctionnels mais offre également des traits impératifs, des traits objet et des traits modulaires. Ce document, réalisé dans le cadre de l'étude LaFoSec, propose une étude des modèles d'exécution du langage OCaml : la compilation et l'exécution sont décrits et analysés du point de vue de la sécurité.

Table des matières

Introduction **7**

1 De la compilation à l'exécution **9**
1.1 Présentation . . . 9
1.2 Phases communes de compilation . . . 12
1.2.1 Préprocesseurs et `camlp4` . . . 14
1.2.2 Analyse statique du texte source . . . 16
1.2.3 Représentation des valeurs en mémoire . . . 18
1.2.4 Comparaison et hachage . . . 25
1.2.5 Constructions invalidant l'analyse statique . . . 29
1.2.6 Compilation séparée, fichiers objet `.cmi` . . . 37
1.2.7 Options de compilation . . . 38
1.3 Édition de liens et chargement . . . 43
1.3.1 Chargement statique . . . 43
1.3.2 Chargement au démarrage par exécutif C . . . 44
1.3.3 Chargement au démarrage par la machine virtuelle . . . 44
1.3.4 Chargement dynamique de code OCaml compilé . . . 44
1.3.5 Interfaçage avec du code en C . . . 48
1.4 Parties communes des exécutifs . . . 48
1.4.1 Traitement des signaux . . . 50
1.4.2 Gestion des *Threads* . . . 51
1.5 Gestion mémoire . . . 51
1.5.1 Allocation, désallocation . . . 52
1.5.2 Ramasse-miettes (GC) . . . 56
1.5.3 GC et données confidentielles . . . 61

2 Modèle d'exécution natif 64
2.1 Compilation . . . 64
2.1.1 Options de compilation . . . 64
2.1.2 Optimisation . . . 65
2.1.3 Compilation séparée, fichiers objet `.cmx`, `.cmxs` et `.cmxa` 66
2.2 Édition de liens et chargement . . . 67
2.2.1 Chargement statique . . . 67
2.2.2 Chargement dynamique de code natif OCaml . . . 67
2.2.3 Interfaçage avec du code C . . . 69
2.3 Exécution et interaction avec le système . . . 69
2.3.1 Traitement des signaux . . . 69
2.3.2 *Backtrace*, pile de levée d'exception . . . 70

3 Modèle d'exécution par machine virtuelle 71
3.1 Compilation . . . 71
3.1.1 Options de compilation . . . 71
3.1.2 Bytecode OCaml . . . 73
3.1.3 Compilation séparée, fichiers objet `.cmo` et `.cma` . . . 75
3.2 Édition de liens et chargement . . . 75
3.2.1 Chargement statique . . . 75
3.2.2 Chargement dynamique de bytecode OCaml . . . 76
3.2.3 Interfaçage avec du code C . . . 78
3.3 Exécution et interaction avec le système . . . 78
3.3.1 Machine virtuelle `ocamlrun` . . . 78
3.3.2 Traitement des signaux . . . 79
3.3.3 Mode débogue . . . 79

4 Modèle d'exécution par boucle interactive 81

5 Interfaçage avec l'environnement système 83
5.1 Variables d'environnement . . . 83
5.2 Privilèges système . . . 84
5.3 Différences d'exécution . . . 85
5.4 Récapitulatifs . . . 86
5.4.1 Extensions de fichiers . . . 86
5.4.2 Options de compilation et de la boucle interactive . . . 87

Bibliographie 88

Table des figures **91**

Table des tables **92**

Table des avis de sécurité **93**
Table des intérêts 93
Table des dangers 94
Table des recommandations 95

Acronymes **98**

Index **99**

Introduction

Objet du document

Ce document a été produit dans le cadre de l'étude LaFoSec, relative au marché n° 2010027960021207501 notifié le 8 novembre 2010 par le Secrétariat Général de la Défense et de la Sécurité Nationale (SGDSN). Il présente les modèles d'exécution du langage OCaml.

Présentation du projet LaFoSec

Les langages de programmation dits *fonctionnels* sont réputés offrir de nombreuses garanties facilitant le développement de logiciels soumis à des exigences de sûreté ou de sécurité. Par exemple, la société *Ericsson* a développé le langage fonctionnel Erlang, dédié à la concurrence, pour ses applications de communication. La société *Esterel Technologies* a développé le langage fonctionnel SCADE, dédié au traitement synchrone de flots de données, pour le traitement de logiciel critique.

Dans le cadre de ses activités d'expertise, l'ANSSI (Agence Nationale de la Sécurité des Systèmes d'Information) souhaite bénéficier d'une assistance scientifique et technique pour déterminer l'adéquation de ces langages au développement d'applications de sécurité et disposer d'une étude permettant d'améliorer la confiance vis-à-vis de la sécurité de ces applications. C'est l'objet du projet LaFoSec.

Le projet LaFoSec consiste en une étude prospective des langages fonctionnels, visant à déterminer les caractéristiques propres à ces langages susceptibles de répondre aux exigences de sécurité.

Tout d'abord, les caractéristiques des langages fonctionnels sont décrites et plusieurs langages généralistes sont présentés selon ces caractéristiques. Ensuite, cette étude est approfondie en se restreignant à trois langages, OCaml, $F^{\#}$ et

Scala, pour recenser leurs avantages et inconvénients du point de vue de la sécurité.

Présentation du contexte

Le langage OCaml est décrit entièrement par son manuel de référence. Il offre des traits de programmation impératifs, objet et fonctionnels et un haut degré de vérification automatique de code par son système de typage et de filtrage (voir les documents [ETAT-LANG, 2011] et [ANA-SECU, 2011] pour plus de détail).

Ce document détaille les modèles d'exécution OCaml et les étudie du point de vue de la sécurité. Le contenu de ce document s'appuie d'une part sur la description du langage faite dans son manuel de référence [Leroy *et al.*, 2010] et d'autre part, sur les travaux de recherche autour de OCaml ainsi que sur des présentations orales ou écrites accessibles en ligne. Ces documents traitent de différents aspects du langage : typage, définition formelle de la sémantique, mécanismes de compilation, ramasse-miettes, etc. En voici une sélection :

- le système d'inférence de types [Pottier et Rémy, 2005] [Garrigue, 2004],
- le système de modules [Leroy, 1994] [Leroy, 1995] [Leroy, 2000],
- les traits objet [Rémy et Vouillon, 1998] [Garrigue et Rémy, 1999] [Leroy, 1999],
- la compilation vers du code natif [Leroy, 1997],
- la machine virtuelle Zinc de OCaml [Leroy, 1990] [Leroy, 2005], et
- le gestionnaire de mémoire [Doligez et Leroy, 1993] [Doligez et Gonthier, 1994] [Doligez, 1995] [Cuoq et Doligez, 2008].

La document fait référence à la version 3.12.0 de OCaml et à sa distribution standard, datant d'août 2010.

Plan du document

Le chapitre 1 présente d'abord le compilateur OCaml et ses trois modèles d'exécution. Il décrit ensuite les parties du compilateur et de l'exécutif communes aux trois modèles. Les chapitres suivants décrivent les particularités de chacun des trois modèles : natif (chapitre 2), par machine virtuelle (chapitre 3) et par boucle interactive (chapitre 4). Le chapitre 5 détaille du point de vue de la sécurité l'interfaçage avec l'environnement système et donne les listes récapitulatives des extensions de fichiers et des options de compilation.

Chapitre 1

De la compilation à l'exécution

Après une présentation générale du compilateur et des modèles d'exécution, ce chapitre décrit les parties du compilateur et de l'exécutif communes aux trois modèles.

1.1 Présentation

Le schéma général de compilation d'un texte source OCaml est donné dans la figure 1.1. Le compilateur prend des textes source OCaml d'extensions `.ml` et `.mli` et produit, selon les options, des fichiers objet OCaml d'extensions `.cmi`, `.cmx`, `.cmxs`, `.cmxa`, `.cmo` et `.cma`. Un texte source `.ml`, appelé **fichier source d'implémentation**, est composé d'une succession de phrases *toplevel* (déclarations, définitions ou expressions à évaluer). Tout fichier source d'implémentation constitue une **unité de compilation** qui correspond à un module au niveau langage. Le rôle de ces différents fichiers sera détaillé dans les sections décrivant la compilation séparée (cf. sections 1.2.6, 2.1.3 et 3.1.3). On peut noter que les formats de ces fichiers objet sont en général conservés au cours des évolutions du compilateur, ce qui permet d'assurer la compatibilité ascendante

Le langage OCaml propose trois modèles d'exécution : code natif, bytecode et boucle interactive. Ils sont brièvement introduits ci-après et seront étudiés dans les chapitres suivants.

Le premier modèle d'exécution de OCaml, détaillé dans le chapitre 2, repose sur l'utilisation du compilateur natif, appelé `ocamlopt`, pour produire du **code natif** (parfois appelé code assembleur ou code machine) qui peut être exécuté directement par le processeur de la machine. Ce code natif est fortement optimisé

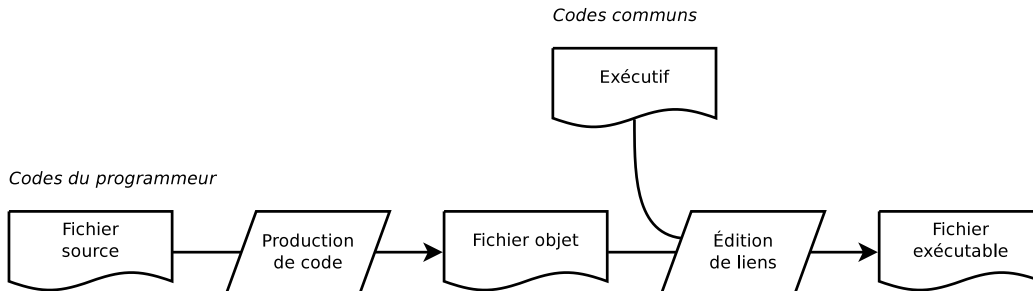


FIGURE 1.1 – Principe général de la compilation

mais non portable : comme tout exécutable natif, il ne peut être exécuté que sur l'architecture pour laquelle il a été compilé.

Le deuxième modèle d'exécution de OCaml, présenté dans le chapitre 3, utilise une machine virtuelle ou interpréteur de bytecode. Le compilateur `ocamlc` produit un **bytecode** à partir de code source OCaml. Ce bytecode peut être ensuite exécuté par la **machine virtuelle** de OCaml dont l'exécutable est nommé `ocamlrun`. Cette machine virtuelle est un programme écrit en C et peut donc être portée sur toute architecture possédant un compilateur C-ANSI. Le bytecode peut donc être exécuté sur n'importe quelle architecture aprés portage de la machine virtuelle sur cette architecture. Ce modèle a donc l'avantage de la portabilité mais conduit à des exécutions moins efficaces qu'avec le modèle natif.

Le troisième modèle d'exécution de OCaml se fait par boucle interactive (chapitre 4). Cette méthode permet l'évaluation interactive d'un code source OCaml en utilisant l'outil dit *toplevel* nommé `ocaml`. Ce mécanisme de Read-Eval-Print Loop (REPL) pour OCaml lit des phrases du langage, les type, les compile vers du bytecode, les évalue (les exécute) et imprime les types et les valeurs résultant de leur évaluation. Chacune des phrases du source, correspondant à une instruction, est soumise à une compilation vers du bytecode, suivie d'une exécution par la machine virtuelle faite dans le contexte défini par la compilation et l'exécution des phrases précédentes. Ainsi ce troisième modèle ne se différencie que peu du deuxième. Dans le chapitre 4, seules les différences entre ces deux modèles sont décrites.

La chaîne de production est presque la même dans les trois modèles : écriture du texte source dans le langage OCaml, compilation avec le compilateur OCaml pour vérifier ce source et production du code natif ou du bytecode et, selon le cas, exécution directe du code natif sur la machine ou utilisation d'outils OCaml pour l'exécution du bytecode. La séparation entre les modèles d'exécution ne se fait que

très tard dans le processus de compilation et cela justifie de parler du compilateur de OCaml au singulier. Cette séparation tardive n'a pas d'inconvénient recensé et présente plusieurs avantages.

- Elle minimise les différences sémantiques entre les différents modèles. Toutes ne peuvent cependant être évitées et ces différences sont recensées en section 5.3. Elles n'ont aucun impact sur la grande majorité des programmes, qui auront donc le même comportement, quel que soit le modèle d'exécution choisi.

  Intérêt I-1: Compatibilité des modèles d'exécution.
  Cette compatibilité permet de développer en mode interactif puis de compiler l'ensemble du développement en étant assuré que le comportement sera le même et qu'aucune faiblesse ne sera introduite par le changement de modèle (voir section 5.3 pour plus de détails). ■

- Cette séparation tardive permet un large partage du code du compilateur entre les différents modèles, le compilateur est donc plus facile à maintenir.
- La communauté d'utilisateurs n'est de ce fait pas scindée en groupes attachés à un modèle donné et contribue de manière cohérente à l'amélioration des outils.

Le compilateur est écrit en OCaml. Son texte source est en fait le premier programme compilé par toute nouvelle version du compilateur (mécanisme appelé **bootstrap**). Ainsi, le compilateur possède lui-même les propriétés garanties par le typage et les analyses statiques décrites dans ce document, ce qui renforce la confiance que l'on peut lui accorder. Ce compilateur est composé d'un **front-end** commun présenté en section 1.2 et de deux **back-end** spécifiques aux modes de compilation :

- le *back-end* natif (cf. section 2.1) associé au *front-end* constitue le compilateur natif `ocamlopt`
- le *back-end* bytecode (cf. section 3.1) associé au *front-end* constitue le compilateur bytecode `ocamlc`.

La compilation d'un fichier source d'implémentation OCaml (extension `.ml`) produit :

- en mode natif, un fichier objet natif d'extension `.o` et un fichier d'annotations d'extension `.cmx`,
- en mode bytecode, un fichier objet OCaml d'extension `.cmo`.

Il existe une seule distribution du compilateur et de la machine virtuelle pour

l'exécution de bytecode OCaml. Cette distribution est faite par l'Institut National de Recherche en Informatique et en Automatique (INRIA), sous licence libre [1]. Le compilateur et les outils de OCaml sont portés sur la majorité des architectures et systèmes d'exploitation. Un même texte source OCaml peut donc être compilé sur toutes ces architectures.

Pour le modèle natif, les architectures maintenues par l'équipe de développement de l'INRIA sont AMD64 (Opteron) sous Linux, MacOS X et MS Windows, IA32 (Pentium) sous Linux, FreeBSD, MacOS X et MS Windows, et PowerPC sous MacOS X. D'autres architectures sont maintenues de manière moins active par l'INRIA ou sont maintenues par la communauté d'utilisateurs (ARM, SPARC, etc.). En ce qui concerne le modèle par machine virtuelle, toute architecture convient dès qu'elle dispose d'un compilateur C-ANSI sous un système d'exploitation POSIX ou Windows. La plupart des distributions Linux contiennent des paquetages OCaml.

Il est à noter que des textes source, écrits dans un noyau OCaml peuvent être compilés par le compilateur du langage F# (inspiré de OCaml) étudié dans [ANA-SECU, 2011], permettant ainsi une exécution sur la plateforme .NET. Cependant, les fichiers objet OCaml ne sont pas exécutables sur la plateforme .NET. Par ailleurs, il existe de nombreux compilateurs et machines virtuelles expérimentaux pour OCaml comme **`Js_of_ocaml`** qui compile du bytecode OCaml en Javascript, **`Ocamljs`** qui compile des textes source OCaml en Javascript, ou **`OCamlJit`** qui est un machine virtuelle pour exécuter du bytecode OCaml par compilation native à la volée JIT (Just-In-Time).

1.2 Phases communes de compilation

Le processus de compilation de OCaml est constitué de passes successives. Chaque passe de compilation produit un **code intermédiaire** [2]. Les premières passes, regroupées sous le nom de **front-end**, sont communes aux trois modèles d'exécution. Il s'agit des passes de **lexing** (analyse lexicale), de **parsing** (analyse grammaticale), de **typage** et de production de **lambda-code** (langage avec un

1. Détails disponibles à la page `http://caml.inria.fr/ocaml/license.fr.html`.
2. Le compilateur enregistre dans des fichiers temporaires certains de ces codes avec le module **`Filename`** qui place ces fichiers dans le répertoire indiqué par la variable d'environnement **`TMPDIR`**, cf. section 5.1.

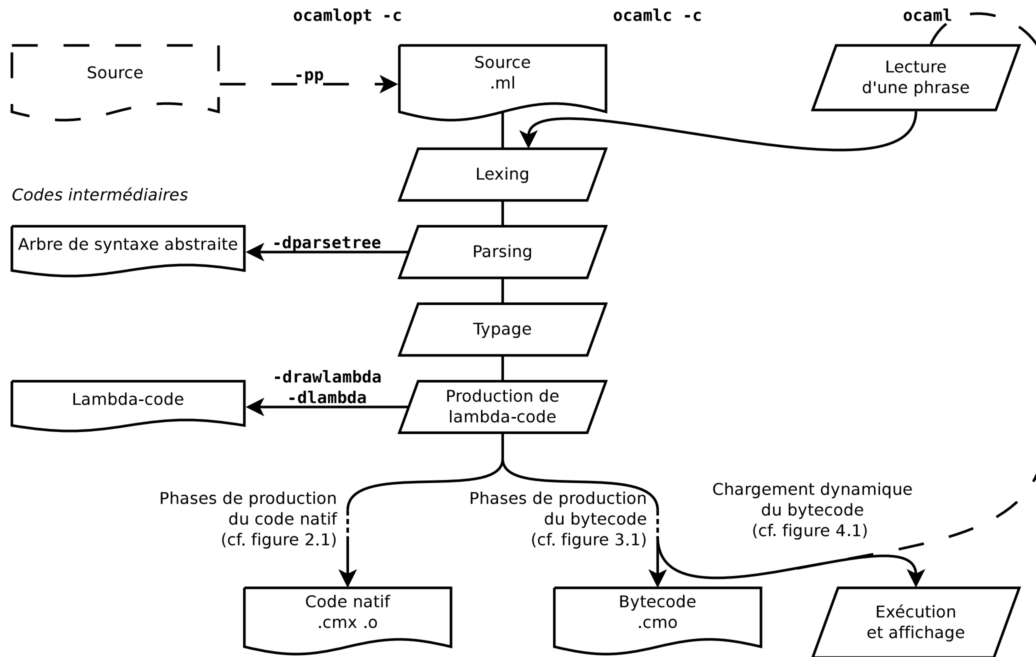


FIGURE 1.2 – Phases communes de compilation

nombre restreint de constructions syntaxiques proche du λ-calcul). Ces phases communes de compilation sont illustrées par la figure 1.2.

Des options du compilateur permettent la mise à disposition des résultats de certaines passes de compilation. Le front-end permet deux types de sortie : un Arbre de Syntaxe Abstraite (AST) résultant du parsing (option `-dparsetree`) et du lambda-code (options `-drawlambda` et `-dlambda`).

> *Intérêt I-2: Mise à disposition des codes intermédiaires produits par le compilateur.*
> Les sorties intermédiaires du compilateur permettent une inspection du code compilé. ■

Le processus de compilation utilise `Filename.tempfile` pour créer des fichiers temporaires placés dans `/tmp` de façon sûre[3].

La phase optionnelle de *preprocessing*, pouvant précéder les passes de lexing et parsing, est discutée en section 1.2.1. Les garanties apportées par la phase de typage sont décrites en section 1.2.2.

3. La sûreté repose sur le mécanisme de protection de `/tmp` sous Unix/Linux.

1.2.1 Préprocesseurs et `camlp4`

Un **préprocesseur** sert à effectuer des manipulations sur le texte source avant sa compilation : à partir d'un texte source, il produit un nouveau texte source. Un tel outil permet de définir des abréviations de texte, une syntaxe appropriée à une application donnée, et permet aussi d'interdire certaines constructions syntaxiques, etc. OCaml n'a pas de préprocesseur interne mais permet de faire appel à un préprocesseur externe avec l'option `-pp`. La distribution OCaml fournit un préprocesseur `camlp4` décrit plus loin.

L'utilisation d'un préprocesseur introduit deux difficultés :

- d'une part, la compilation du texte issu du *preprocessing* détecte les erreurs sur ce dernier source et ne peut pas, en général, localiser l'erreur sur le texte original, ce qui rend la correction difficile ;
- d'autre part, le préprocesseur peut introduire du texte source malveillant dans le texte original.

> *Danger D-1: Introduction de source malveillant par l'utilisation de préprocesseurs.*
> Le *preprocessing* peut modifier le texte source original, de manière non détectable si ce n'est par lecture du texte issu du *preprocessing*. ■

> *Recommandation R-1: S'assurer de l'innocuité d'un préprocesseur avant son utilisation.*
> Si l'utilisation d'un préprocesseur est requise, il est conseillé de faire une analyse des transformations qu'il opère ou de vérifier le texte résultat du *preprocessing* transmis au compilateur OCaml, en produisant une sortie intermédiaire. Toute utilisation de l'option `-pp` du compilateur doit être justifiée. ■

Le *preprocessing* est effectué en utilisant l'option de compilation `-pp`. Cette option prend en argument une commande qui, appliquée au texte source à modifier, produit soit un texte source OCaml soit directement un AST OCaml. Dans le premier cas, le processus de compilation effectue la phase de *lexing-parsing* sur le source résultat, puis construit l'AST et effectue son typage. Dans le second cas, la phase de *lexing-parsing* n'est pas nécessaire, et le compilateur procède directement au typage de l'AST produit.

Il est à noter que le préprocesseur est exécuté avec les mêmes droits que le compilateur, voir la section 5.2.

OCaml, macros

Le mécanisme d'expansion des macros du langage C est un exemple connu de préprocesseur interne. Une **macro** permet de remplacer un élément syntaxique par du texte source (`#define` du langage C). Ce mécanisme peut rendre difficile la localisation des erreurs remontées par la compilation. Le langage OCaml ne possède pas de mécanisme de définition et d'expansion de macros (préprocesseur interne). L'absence de macros maintient la proximité entre la syntaxe concrète du langage et le texte analysé par le compilateur.

L'outil `camlp4`

La distribution OCaml fournit l'outil `camlp4` qui permet de définir un préprocesseur adapté à une application (voir [OUTILS-OCAML, 2011] pour une étude de cet outil). Il est à noter qu'une version de `camlp4`, développée en parallèle, est actuellement maintenue hors de la distribution officielle sous le nom de `camlp5`.

Les fonctionnalités principales de l'outil `camlp4` sont :

– l'extension ou la restriction du langage OCaml sans modification du compilateur lui-même par l'ajout ou le retrait de constructions syntaxiques au *parser* OCaml extensible de `camlp4`.
– la manipulation simple et sûre de données complexes par l'utilisation de *quotations* dans le code source. Le *parser* OCaml rencontrant une *quotation* (délimitée par des chevrons << >>), appelle le *parser* dédié, défini à partir d'une syntaxe concrète donnée par le développeur.

Recommandation R-2: Préférer `camlp4` *comme préprocesseur.*

L'utilisation des extensions de syntaxe ou des *quotations* `camlp4` est recommandée si un *préprocessing* est nécessaire pour les raisons suivantes.

– La transformation effectuée par `camlp4` produit un AST, qui est une valeur OCaml de type `Camlp4.PreCast.Ast`. Tout AST produit est donc garanti d'être bien formé. Il peut cependant être mal typé, ce qui sera diagnostiqué par le typage de l'AST.
– L'AST produit par `camlp4` possède des annotations de localisation équivalentes à celles du compilateur seul. Ainsi, les erreurs diagnostiquées par la compilation de cet AST peuvent être facilement localisées dans le code source original.
– Le préprocesseur peut produire, à partir de l'AST produit, un texte source (`.ml`) qui peut donc être soumis à relecture, répondant ainsi au danger

D-1.

Intérêt I-3: L'outil **`camlp4`** *peut servir à analyser le texte source.*
L'outil **`camlp4`** peut de fait être utilisé comme analyseur de code source traquant des constructions non-sûres (voir danger D-13) par restriction de la syntaxe acceptable. On peut de même interdire les différentes formes d'affectation en les enlevant de la syntaxe.■

Recommandation R-3: Utiliser **`camlp4`** *pour adapter la syntaxe de chaque fichier* `.ml` *à son niveau de sécurité.*
Il est possible de spécialiser le *parser* **`camlp4`** selon les propriétés requises sur le fichier. On peut par exemple, interdire l'affectation ou indiquer quelles sont les seules fonctions utilisables dans le texte source à traiter. Le mécanisme de compilation séparée permet de traiter les différents textes source d'une application en choisissant une spécialisation de **`camlp4`** adaptée pour chaque fichier. ■

1.2.2 Analyse statique du texte source

L'analyse statique du texte source effectue en une passe le typage et plusieurs analyses sémantiques portant sur le filtrage, l'utilisation des tableaux et chaînes de caractères, etc. Cette passe est communément appelée "typage" du texte. Les analyses effectuées durant cette phase (analyse de la portée des identificateurs, calcul des informations de typage, etc.) apportent des garanties fortes sur le code source.

Le compilateur instrumente le code de manière à vérifier dynamiquement le non débordement des tableaux et des chaînes de caractères et la non division par zéro.

Ces garanties sont énumérées dans cette section, et détaillées dans [ANA-SECU, 2011]. Les constructions les contournant sont décrites dans la section 1.2.5.

Garanties apportées par le typage

Intérêt I-4: Le typage facilite le développement d'applications.
OCaml n'offre aucune possibilité de manipulation directe de pointeurs. Le processus de compilation garantit que tous les accès à la mémoire sont corrects (il ne peut pas y avoir de débordement de tableaux, ni de pointeur invalide). ■

Intérêt I-5: Le typage garantit de nombreuses propriétés du texte source.

La phase de typage apporte les garanties suivantes sur le texte source :

- Absence de données manipulées non initialisées.
- Impossibilité d'accès à un identificateur en dehors de sa portée et donc validation du cloisonnement des lectures-écritures au niveau source.
- Cohérence de la manipulation des données vis-à-vis de leurs types (impossibilité d'erreurs de segmentation).
- Impossibilité de modifier les liaisons des identificateurs. Les valeurs des constantes ne sont pas modifiables. Seules les valeurs référencées (ou les valeurs de champs mutables) sont modifiables, les références elles-mêmes (ou les champs mutables) ne l'étant pas.
- Accès contrôlé aux éléments des tableaux et chaînes de caractères (instrumentation du code vérifiant, à l'exécution, le non-dépassement des bornes).
- Protection des données de types abstraits et de l'encapsulation (non violation des invariants).
- Analyse de l'exhaustivité du filtrage (identification des filtrages de motifs non complets). ■

Intérêt I-6: Le processus de compilation garantit la préservation des propriétés du texte source.

Une expression contenant un identificateur hors de portée (i.e. non présent dans l'environnement de cette expression) est rejetée par le typage. Dans un code OCaml bien typé, il est donc impossible d'accéder à une variable locale hors de sa portée. De même, il est impossible d'exploiter la représentation d'une valeur d'un type abstrait. Si un module exporte un type abstrait, la définition de ce type n'est pas visible hors de son unité de compilation, les valeurs de ce type ne peuvent être manipulées que par les fonctions d'accès exportées par ce module.

En revanche, les informations de typage ne sont pas conservées dans le code compilé, que ce soit du code natif ou du bytecode. Aucune vérification de typage ne peut être faite directement sur du code compilé. Il n'y a pas de typage dynamique en OCaml.

Danger D-2: Les informations de typage ne sont pas incluses dans

le code compilé.

Des vérifications dynamiques comme celle du non dépassement des bornes sont ajoutées par le compilateur et sont effectuées à l'exécution.

> *Intérêt I-7: Le compilateur insère des vérifications dynamiques de propriétés dans le code compilé.*

> *Recommandation R-4: Ne créer un exécutable qu'à partir de textes source.*
> Il est recommandé de ne pas inclure de fichiers objet ou de bibliothèques fournies sous forme binaire par une tierce partie. Il faut créer un exécutable à partir des textes source et vérifier les commandes de compilation afin que la vérification de typage soit ainsi appliquée à l'ensemble du développement. Les garanties apportées par le typage n'étant portées que par le texte source et non par le code compilé, il est important d'échanger du texte source plutôt que du code compilé. ■

1.2.3 Représentation des valeurs en mémoire

Le propos de cette section est de décrire la représentation en mémoire des données manipulées par le programme. Cette section est un préalable à l'étude de la comparaison et du hachage de la section 1.2.4 suivante.

OCaml effectue conjointement l'allocation et l'initialisation en mémoire des valeurs. La représentation en mémoire de ces valeurs ne sont pas visible par le programmeur OCaml. Elle est structurée en **mots de la machine** de 32 bits ou 64 bits selon l'architecture. Une suite de mots participant à la représentation d'une valeur est appelée un **bloc mémoire**.

Il n'y a pas de marque associée aux valeurs en mémoire témoignant de leur type, de leur caractère mutable ou non et de leur encapsulation. C'est l'algorithme de typage qui garantit lors de la compilation que les manipulations de ces valeurs seront faites conformément à leurs propriétés. Il n'y a pas de mécanisme de cloisonnement de la mémoire à proprement parler en cours d'exécution.

Entiers, caractères

Les entiers signés (de type atomique `int`) sont représentés par un mot dans lequel le bit de poids faible est à `1` et le bit de signe est le bit de poids fort. Cela signifie que l'intervalle des entiers représentables va de -2^{30} à $2^{30}-1$ pour les processeurs 32 bits et -2^{62} à $2^{62}-1$ pour les processeurs 64 bits. Ces bornes sont données par les identificateurs `min_int` et `max_int`. Le compilateur émet une erreur de compilation quand le texte source contient un littéral entier sortant de cet intervalle mais il n'y a pas de levée d'exception en cas de **dépassement d'entier** ou *overflow* par une opération arithmétique à l'exécution.

> *Danger D-3: Pas de détection de dépassement d'entier à l'exécution.*

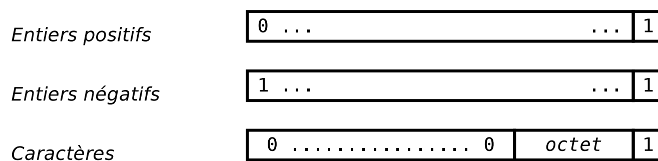
> *Recommandation R-5: Ne pas oublier que l'arithmétique des entiers est une arithmétique modulaire.*

L'exemple suivant illustre le dépassement d'entier sur un processeur 64 bits ($2^{62} = 4611686018427387904$).

```
1 max_int ;;
2 - : int = 4611686018427387903
3 4611686018427387904 ;;
4 - : int = -4611686018427387904
5 4611686018427387905 ;;
6 Error: Integer literal exceeds the range of ↵
    representable integers of type int
7 max_int + 1 ;;
8 - : int = -4611686018427387904
```

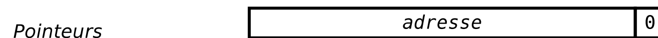
Les caractères (i.e. `'a'` ou `'\097'`) sont représentés comme des entiers entre 0 et 255.

Les caractères accentués ISO 8859-1 sont acceptés dans le texte source OCaml. Il est déconseillé de les utiliser car ce texte source n'est pas adapté à un environnement international.



Pointeurs

Les **pointeurs**, manipulables uniquement par l'exécutif OCaml, sont représentés par un mot, dont le bit de poids faible est `0`, ce qui les différencie des entiers. Cela permet de distinguer facilement pointeurs et entiers et d'optimiser le GC. En effet, les entiers sont directement accessibles dans les registres, contrairement aux autres structures de données qui utilisent un pointeur.



Blocs

Les pointeurs dans le tas pointent vers un **bloc** avec un mot d'en-tête composé de 8 bits d'étiquette ou *tag*, 2 bits pour le GC, et les bits restants ($n - 10$ bits soit 22 ou 54 selon le processeur) pour la taille c'est-à-dire le nombre de champs du bloc (le nombre maximum de champs est donc $2^{22} - 1$ en 32 bits et $2^{54} - 1$ en 64 bits)[4]. Chaque champ est représenté par un mot. Le GC utilise les étiquettes de blocs pour savoir si le bloc peut contenir des pointeurs. Les deux bits réservés au GC lui servent à gérer la mémoire. Les étiquettes servent également au filtrage car elles représentent les constructeurs des types somme.

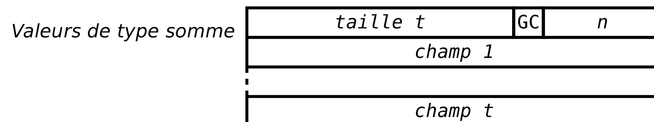
Les pointeurs hors du tas OCaml sont utilisés par le système OCaml pour représenter certaines valeurs particulières comme le vecteur vide, pour donner accès à des portions de code OCaml (dans les fermetures ou les tables de méthodes) ou à des structures de données provenant de C par utilisation de `external`.

Valeurs de types somme, listes, options, booléens et `unit`

La représentation d'une valeur d'un type somme dépend de son constructeur. OCaml attribue un numéro à chaque constructeur. Les types `list`, `option`, `bool` et `unit` sont des types somme particuliers. La limite du nombre de constructeurs est 246 qui est le nombre d'étiquettes de bloc différentes. Une valeur réduite à

4. Ce codage permet la création de blocs de taille nulle mais ceux-ci ne sont pas utilisés dans le tas.

un constructeur constant (i.e. qui n'a pas d'argument) est représentée par un entier. Une valeur construite avec un constructeur possédant des arguments est représentée par un bloc dont l'étiquette est le numéro du constructeur et la taille est le nombre d'arguments de ce constructeur. Chaque argument est représenté par un champ du bloc.

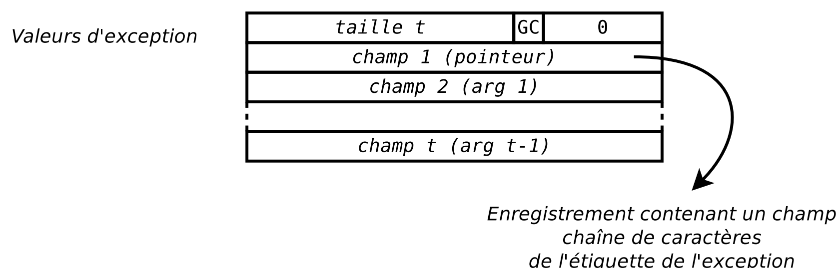


Enregistrements, références, produits et tableaux

Une valeur d'un enregistrement est représentée par un bloc d'étiquette 0 dont la taille correspond au nombre des champs de l'enregistrement. Chaque champ de l'enregistrement est placé dans un champ du bloc, suivant l'ordre d'énonciation de l'enregistrement dans le texte source. Le type `ref` est un enregistrement ayant un champ unique `contents`. Les modules, les produits (`*`) et les tableaux (`array`) sont représentés comme des enregistrements. Les produits OCaml sont n-aires. La limite du nombre de champs d'un enregistrement, d'un tableau ou d'éléments d'un produit dépend de la limite de taille d'un bloc et donc de l'architecture, elle est de $2^{22} - 1$ en 32 bits et $2^{54} - 1$ en 64 bits.

Valeurs d'exception

Les valeurs de type `exn`, dont le constructeur possède n arguments, sont représentées comme un enregistrement par un bloc d'étiquette 0 et de $n + 1$ champs. Le premier champ est un pointeur vers un enregistrement contenant une chaîne de caractères représentant le nom (ou étiquette) de l'exception. Les autres champs correspondent aux arguments.



Le rattrapage d'exception ou le filtrage de valeurs d'exception s'effectue en comparant l'adresse du pointeur de la valeur d'exception avec celle du filtre.

L'exemple suivant définit deux exceptions ayant la même étiquette `E` (lignes 1 et 9). Le filtrage de valeurs d'exception compare les adresses des pointeurs et différencie donc les deux valeurs (la première exception est filtrée par la ligne 3 alors que la deuxième est filtrée par la ligne 4). La comparaison ne distingue pas les deux valeurs d'exception (comme le montre le test de la ligne 4), ce comportement de la comparaison des exceptions est discuté en section 1.2.4 et son impact sur la sécurité dans [ANA-SECU, 2011].

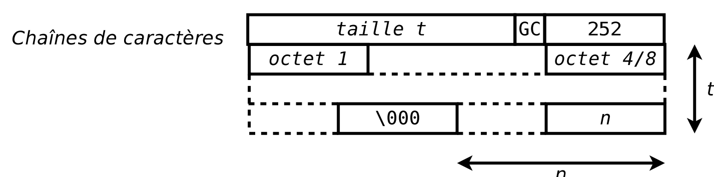
```
1  exception E;;
2  let filtre_E = function
3    | E -> "E"
4    | exc when exc = E -> "Autre E"
5    | _ -> "Autre exception";;
6  val filtre_E : exn -> string = <fun>
7  filtre_E E;;
8  - : string = "E"
9  exception E;;
10 filtre_E E;;
11 - : string = "Autre E"
```

Recommandation R-6: Interdire le masquage des exceptions.
Il est recommandé d'interdire le masquage des exceptions car il peut mener à un contournement de l'encapsulation (voir [ANA-SECU, 2011]). ■

Chaînes de caractères

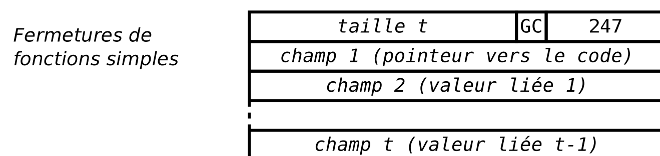
Les chaînes de caractères sont représentées par un bloc d'étiquette 252 avec 4 ou 8 caractères par champ suivant l'architecture (les champs étant composés de

4 octets en 32 bits et de 8 octets en 64 bits). Le compilateur ajoute le caractère 0 (\000) en fin de chaîne. Le nombre de champs d'un bloc représentant une chaîne de caractères de longueur l est donc $(l+1) \div 4$ ou $(l+1) \div 8$. Dans le dernier octet du dernier champ du bloc, le compilateur indique le nombre d'octets entre ce caractère 0 et la fin de la chaîne[5]. La longueur d'une chaîne représentée par n champs et dont le dernier octet du dernier champ a pour valeur m est donc $n * 4 - m - 1$ ou $n * 8 - m - 1$. Ainsi, la longueur maximum d'une chaîne est de $(2^{22} - 1) * 4 - 1$ caractères en 32 bits et $(2^{54} - 1) * 8 - 1$ en 64 bits.



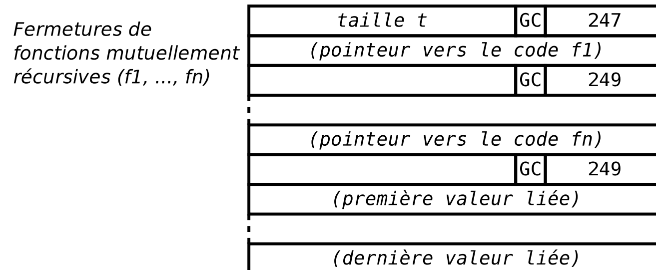
Fonctions, fermetures et fonctions récursives

Une fermeture simple, c'est-à-dire celle d'une fonction non récursive, est un bloc d'étiquette 247 ayant comme premier champ un pointeur vers le code de la fonction. Les autres champs correspondent aux valeurs liées par la fermeture. La taille du bloc dépend donc du nombre de liaisons faites par la fermeture.



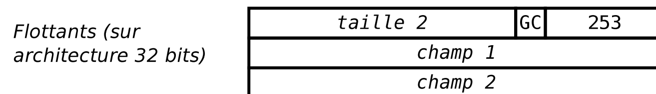
Une fermeture de fonctions mutuellement récursives est également représentée par un bloc d'étiquette 247. Ce bloc contient, pour chaque fonction récursive, un champ contenant un pointeur vers le code de la fonction et un champ ressemblant à une en-tête de bloc d'étiquette 249. Ensuite viennent les valeurs liées par la fermeture, une par champ.

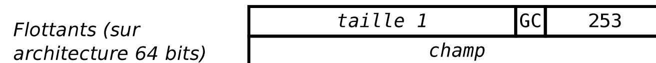
5. Si l'emplacement du caractère 0 coïncide avec le dernier octet du dernier champ, celui-ci représente bien le nombre d'octets entre ce caractère 0 et la fin du dernier champ.



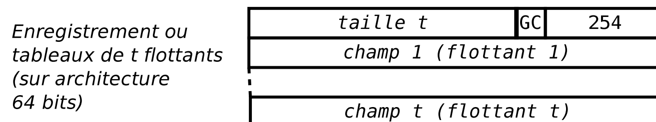
Flottants

Les flottants sont représentés par un bloc d'étiquette 253 contenant deux champs en 32 bits et un seul en 64 bits.



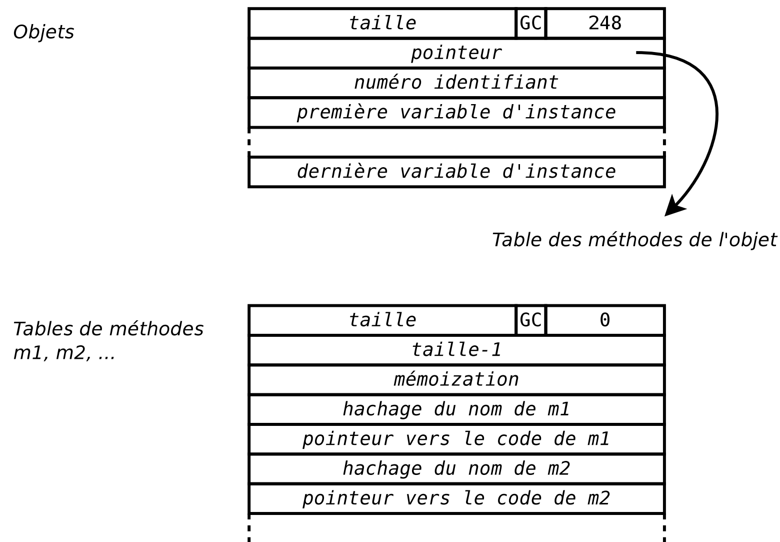


Un enregistrement ou un tableau de flottants est représenté d'une manière particulière qui réduit significativement le nombre d'instructions requises pour leur manipulation. Ils sont représentés par un bloc d'étiquette 254 ayant deux champs par flottant en 32 bits et un champ par flottant en 64 bits (contre un pointeur et un bloc flottant en représentation habituelle des enregistrements et des tableaux).



Objets et méthodes

Un objet est représenté par un bloc d'étiquette 248 dont le premier champ est un pointeur vers la table de ses méthodes, le deuxième champ est l'identificateur de l'objet et les champs suivants sont ses variables d'instance. Une table de méthodes est représentée par un bloc dont le premier champ représente la longueur de la table, le deuxième champ est réservé à la mémoïsation, et des paires de champs représentant un hachage du nom de la méthode et un pointeur vers son code.



1.2.4 Comparaison et hachage

Cette section étudie la comparaison et le hachage des données, à partir de la description de leur représentation en mémoire.

Comparaison

Le langage OCaml propose des opérations (`=`, `<>`, `<`, `>`, `<=` et `>=`) et une fonction (`compare`) de comparaison. Les comparaisons ne sont appelées que sur des valeurs de même type sauf en cas de contournement du typage (cf. section 1.2.5). Ces **comparaisons de valeurs** sont effectuées sur la représentation interne de ces valeurs par une fonction C (`compare_val`) selon deux modes différents : le mode standard pour les opérations d'égalité et de comparaison et le mode total pour la fonction `compare`. Le mode total a un comportement différent vis-à-vis :

– des flottants (`nan` est égal à lui même)

```
nan = nan;;
- : bool = false
compare nan nan;;
- : int = 0
```

– du parcours de la structure (le mode total s'arrête si deux adresses sont identiques sans parcourir la structure interne de la valeur). Ainsi la fonction `compare` ne lève pas toujours une exception lors de la comparaison de fonctions au sein d'une sous-structure. En effet la fonction `compare` retourne `0` si les sous-structures comparées sont des pointeurs vers la même fermeture alors qu'elle lève l'exception `Invalid_argument "equal: functional value"` si les pointeurs diffèrent.

L'exemple suivant illustre les différences de comportement de l'égalité (`=`), de l'égalité physique (`==`) et de la fonction `compare` sur des enregistrements ayant une valeur fonctionnelle.

```
module M : sig
  type t
  val f : t val g : t val h : t
end = struct
  type t = { c : int -> int }
  let id = (fun x -> x)
  let f = { c = id }
  let g = { c = id }
  let h = { c = (fun x -> x) }
end;;
module M : sig type t val f : t val g : t val h : t end
M.f = M.g;;
Exception: Invalid_argument "equal: functional value".
M.f = M.h;;
Exception: Invalid_argument "equal: functional value".
M.f = M.f;;
Exception: Invalid_argument "equal: functional value".
M.f == M.g;;
- : bool = false
M.f == M.h;;
- : bool = false
M.f == M.f;;
```

```
23 - : bool = true
24 compare M.f M.g;;
25 - : int = 0
26 compare M.f M.h;;
27 Exception: Invalid_argument "equal: functional value".
28 compare M.f M.f;;
29 - : int = 0
```

Trois valeurs sont définies : `f` et `g` dont la valeur fonctionnelle est égale à la même fonction `id` et `h` dont la valeur fonctionnelle est une fonction anonyme égale à celle de `id`. Il est à noter que ces trois valeurs sont de type `t` (ligne 2) encapsulé dans le module `M`. L'opération d'égalité entre ces valeurs (mode standard de `compare_val`) n'aboutit pas du fait du champ fonctionnel de l'enregistrement (lignes 12–17). L'opération d'égalité physique teste l'égalité des pointeurs des identificateurs (lignes 18–23) et ne renvoie `true` que pour le troisième test. La fonction `compare` (mode total de `compare_val`) parcourt les structures des valeurs et teste les pointeurs des valeurs fonctionnelles (lignes 24–29). Ainsi, `f` et `g` sont reconnus égaux (ligne 25) car partageant la valeur fonctionnelle liée à l'identificateur `id`. Il est à noter que cette comparaison contourne l'encapsulation du module `M` (cf. section 1.2.5).

La fonction C `compare_val` de la bibliothèque standard OCaml effectue une comparaison machine des entiers. Si la comparaison est faite entre un entier et un bloc, l'entier est systématiquement considéré plus petit (ceci a une incidence sur l'ordre des valeurs de type somme, voir plus bas). La comparaison de pointeurs hors du tas est une comparaison de leurs adresses. En mode total, la comparaison de pointeurs retourne 0 si les pointeurs sont égaux. Si les pointeurs ne sont pas égaux ou si le calcul se fait en mode standard, la comparaison de pointeurs du tas effectue une comparaison des valeurs pointées.

> *Danger D-4: L'exécution de la comparaison n'est pas typée.*
> En contournant statiquement les protections du typage (cf. section 1.2.5), il est possible de comparer des valeurs de types différents car il n'y a pas de vérification dynamique du typage. ■

> *Danger D-5: La comparaison de valeurs d'un type somme n'est pas complètement spécifiée.*
> Le comportement de la comparaison de valeurs d'un type somme n'est pas spécifié et peut changer dans les futures versions du compilateur. La comparaison entre valeurs d'un type somme se base sur

l'ordre des constructeurs dans la définition du type mais toutes les valeurs de constructeurs constants (constructeurs sans argument) sont inférieures aux valeurs de constructeurs paramétrés. ■

Recommandation R-7: N'utiliser les fonctions de comparaison génériques que si la définition précise de l'ordre n'a pas d'importance.
Dès qu'une application utilise certaines caractéristiques supplémentaires de l'ordre, il est nécessaire de définir ses propres fonctions de comparaison. ■

La comparaison entre exceptions s'effectue, quant à elle, à l'exécution et donc sans connaissance du type `exn` de la valeur (la représentation d'une exception est celle d'un enregistrement). La comparaison s'effectuant par récursion sur la représentation, elle effectue une simple comparaison des noms des exceptions. Elle permet donc la comparaison d'exceptions de même nom, mais avec un nombre et des types d'arguments différents.

Danger D-6: Possibilité de contournement du typage et de l'encapsulation par la comparaison d'exceptions.
Ce danger est expliqué dans [ANA-SECU, 2011]. ■

Recommandation R-8: Examiner l'utilisation des exceptions dans tout texte source non digne de confiance.
Il est nécessaire de contrôler toute utilisation d'une exception afin de détecter toute comparaison malveillante. ■

Hachage

Le module `Hashtbl` propose deux fonctions de hachage de valeurs, `hash` et `hash_param`. Ces fonctions de hachage retournent une empreinte sous la forme d'un entier naturel compris entre 0 et $2^{30} - 1$ quelle que soit l'architecture.

Pour les entiers, le hachage correspond à une identité modulo 2^{30} (et donc l'identité pour les caractères). Pour les pointeurs hors du tas, l'empreinte est la valeur de l'adresse modulo 2^{30}. Le calcul du hachage d'un bloc se fait récursivement sur sa structure.

Danger D-7: Le hachage ne connaît pas les limites de l'abstraction.
Le hachage, se faisant à l'exécution, ne tient pas compte du type des valeurs ni des encapsulations, il permet donc de les contourner (cf. section 1.2.5). ■

1.2.5 Constructions invalidant l'analyse statique

Certaines constructions du langage peuvent briser la protection apportée par le typage et sont dites ici **non-sûres**. Les constructions non-sûres de la bibliothèque standard sont regroupées ci-dessous selon le contournement qu'elles permettent.

Constructions permettant de contourner le typage :

- les fonctions du module `Obj`,
- les fonctions de sérialisation/désérialisation du module `Marshal`,
- les fonctions déclarées par la construction `external`.

Constructions contournant la vérification des bornes à l'exécution :

- l'option `-unsafe`,
- les fonctions `unsafe_*` des modules `Pervasives` (importé par défaut), `Array`, `ArrayLabels`, `Bigarray`, `Char`, `StdLabels`, `String`, `StringLabels` et `Printf.CamlinternalPr.Sformat`.

Construction permettant la récupération de valeurs désallouées :

- la fonction `create` du module `String`.

Constructions permettant de contourner l'encapsulation :

- les fonctions du module `Printexc`,
- les opérations d'égalité (`=`, `<>`) du module `Pervasives`,
- les opérations et fonctions de comparaison (`<`, `>`, `<=`, `>=`, `compare`) du module `Pervasives`,
- les fonctions de hachage du module `Hashtbl`.

Module `Obj`

Le contournement du typage s'avère parfois nécessaire au cours de la définition de traitements de très bas niveau ou au cours du traitement de fichiers typés à l'aide de systèmes de types plus puissants que celui de OCaml, les fichiers source extraits de Coq par exemple.

Les fonctions `obj` et `magic` du module `Obj` permettent d'attribuer un type quelconque à toute valeur et brisent ainsi la protection apportée par le typage, le programme restant accepté par le compilateur. Les fonctions du module `Obj` permettent aussi d'accéder à des variables d'instance d'un objet et à ses méthodes privées cachées en forçant le type de cet objet.

> *Danger D-8: L'utilisation du module* `Obj` *détruit la cohérence du texte source apportée par le typage.*

Recommandation R-9: Ne pas utiliser le module `Obj`.
Le préprocesseur **camlp4** peut être utilisé pour outiller cette recommandation. ■

Recommandation R-10: S'assurer de la non-utilisation du module `Obj` *par les bibliothèques externes chargées.*
Cela ne peut être fait que par vérification des sources des bibliothèques. ■

Recommandation R-11: Justifier et confiner toute utilisation du module `Obj`.
Si l'utilisation du module `Obj` est indispensable, il est recommandé de confiner et de contrôler cette utilisation en l'encapsulant dans un module. ■

Module `Marshal`

Le module `Marshal` propose des fonctions de sérialisation et désérialisation pour la communication ou le stockage de valeurs OCaml. Une valeur est sérialisée (fonctions `Marshal.to_*`) sous forme binaire pour être enregistrée dans un fichier ou transmise sur un canal de communication. Seule sa valeur est enregistrée, son type ne l'est pas. La désérialisation (fonctions `Marshal.from_*`) se fait donc sans vérification de type. Une valeur sérialisée ne portant pas son propre type, celui-ci doit être fourni explicitement par le développeur au moment de la désérialisation : `(Marshal.from_*:`*type*`)`. Il est à noter que les données sérialisées par le module `Marshal` sont fournies sous forme binaire non chiffrée, elles restent donc facilement lisibles.

Danger D-9: La sérialisation/désérialisation peut donner accès à la zone mémoire utilisée par le programme.

Recommandation R-12: Définir des opérations robustes pour la transmission de données.
Définir un *parser* et un *printer* adaptés au type des données à transmettre, remplaçant les opérations de sérialisation et désérialisation. La distribution OCaml propose deux générateurs de *parsers* : **ocamllex/ocamlyacc** et **camlp4**. Il existe également un

générateur de *parsers* **menhir**[6] qui est destiné à se substituer à **ocamlyacc**. Le typage mis en œuvre par ces outils garantit que les valeurs qu'ils produisent sont bien du type requis.

De plus, le format de communication des données peut être choisi de façon à faciliter différentes formes de contrôle (par d'autres outils que OCaml éventuellement). ■

Recommandation R-13: Éviter d'utiliser le module `Marshal`*.*
Sauf obligation par la spécification, l'utilisation du module `Marshal` est à proscrire. Si on doit l'utiliser, il faut contrôler l'origine de la valeur à désérialiser et s'assurer qu'elle a bien le type indiqué pour éviter une corruption du programme. On peut pour cela utiliser une signature cryptographique de la donnée sérialisée. ■

La sérialisation/désérialisation des données construit une représentation d'une valeur structurée en concaténant les différents blocs constituant la structure. Les pointeurs vers le tas présents dans la structure sont remplacés par le bloc vers lequel ils pointaient. Une valeur dont la structure utilise un pointeur hors tas ne peut pas être sérialisée, sauf si la fonction `Marshal.to_*` a reçu l'option `Marshal.Closures`. Dans ce dernier cas, les pointeurs de code sont acceptés et sont associés à un hachage MD5 du code du programme qui permet de vérifier leur cohérence au cours de la désérialisation. Le mécanisme de hachage MD5 ne protège cependant pas contre la modification délibérée des valeurs sérialisées. Par exemple, un pointeur de code peut être remplacé par un autre pointeur de code et la valeur sérialisée pourra cependant être désérialisée conformément au type attendu.

Danger D-10: Aucune vérification n'est faite sur les valeurs à désérialiser.

Cependant, il n'est pas possible de charger du code par l'intermédiaire d'une désérialisation. Celle-ci ne crée que des valeurs dans le tas et ne crée pas du code (pour plus de précisions sur la protection du code voir la section 1.5.1). La structure construite par désérialisation peut contenir un pointeur de code à une adresse arbitraire : si l'adresse est dans la zone de code elle doit pointer sur du code pré-existant, si l'adresse est hors de la zone de code la protection éventuellement apportée par le système d'exploitation peut empêcher l'exécution de ce code.

6. `http://gallium.inria.fr/~fpottier/menhir/`

En résumé, l'utilisation du module `Marshal` peut rompre la cohérence du typage, rompre l'encapsulation mais ne fait qu'allouer de nouvelles valeurs dans le tas.

Dans l'exemple suivant, une valeur encapsulée `Secret.v` (définie à la ligne 7) d'un type abstrait `Secret.t` est sérialisée (ligne 10). Sa désérialisation (ligne 12) retourne une valeur polymorphe. L'attribution de types concrets à cette valeur polymorphe peut aboutir à des valeurs concrètes (lignes 14–20) et révéler la valeur initialement encapsulée si la structure du type attribué équivaut à celle du type abstrait (lignes 21–26). Ceci illustre le fait que sérialisation et désérialisation contournent l'encapsulation. Une erreur de segmentation peut arriver si la représentation interne du type attribué n'est pas compatible avec celle de la valeur sérialisée (lignes 27–29).

```
1  module Secret :
2      sig
3        type t
4        val v : t
5      end = struct
6        type t = S of string * int
7        let v = S ("secret", 12345)
8      end;;
9  module Secret : sig type t val v : t end
10 let v_out = Marshal.to_string Secret.v [];;
11 val v_out : string = ↵
       "\132\149¦¾\000\000\000\011\000\000\000\002\000\000 ↵
       \000\006\000\000\000\005_&secret\00109"
12 let v_in = Marshal.from_string v_out 0;;
13 val v_in : 'a = <poly>
14 let v_in_int : int = v_in;;
15 val v_in_int : int = 2149197136
16 let v_in_string : string = v_in;;
17 val v_in_string : string = ↵
       "¸J4\000\001\000\000\000s`\000\000\000\000\000"
18 type essai1 = int * int;;
19 let v_in_essai1 : essai1 = v_in;;
20 val v_in_essai1 : essai1 = (2149197148, 12345)
21 type essai2 = { s : string; i : int };;
22 let v_in_essai2 : essai2 = v_in;;
23 val v_in_essai2 : essai2 = {s = "secret"; i = 12345}
24 type essai3 = Essai3 of string * int;;
```

```
25 let v_in_essai3 : essai3 = v_in;;
26 val v_in_essai3 : essai3 = Essai3 ("secret", 12345)
27 type essai4 = string * string;;
28 let v_in_essai4 : essai4 = v_in;;
29 segmentation fault
```

Construction `external`

OCaml offre un mécanisme d'interfaçage avec du code natif. Ce code peut être obtenu par compilation à partir de n'importe quel langage mais est le plus souvent issu d'un source en C. La construction `external` permet de déclarer une fonction dont l'appel lancera l'exécution d'une fonction C donnée. Par exemple, `external fn_ocaml : int -> int = "fn_c"` déclare une fonction `fn_ocaml` de type OCaml `int -> int`. L'appel de `fn_ocaml` exécutera la fonction `fn_c` en lui passant son paramètre effectif. Le type de la fonction est donné explicitement par le programmeur. Le chargement du code externe diffère selon le mode d'exécution et est décrit dans les sections 2.2.3 et 3.2.3.

Les accès mémoire par le code C appelé dans un programme OCaml se font dans l'espace mémoire de ce programme OCaml. Ce code C peut lire et écrire dans cet espace et donc modifier arbitrairement les données du programme OCaml. Le programme OCaml ne peut pas accéder à l'espace utilisé par le programme C, sauf s'il utilise des constructions non-sûres.

> *Danger D-11: Les accès à la mémoire effectués par du code externe ne sont pas contrôlés.*

> *Recommandation R-14: Éviter l'utilisation de la construction* `external`.
> L'utilisation de bibliothèques contenant du code C est à éviter s'il n'est pas possible de faire confiance à ce code. Le compilateur OCaml n'a pas la possibilité de vérifier ni d'intercepter les accès mémoire provenant du code C. ■

> *Recommandation R-15: Analyser les sources C avant leur utilisation.*
> Si l'utilisation de code C s'avère indispensable, il est recommandé de le vérifier par relecture du texte source ou bien, s'il provient d'un

fournisseur de confiance, de s'assurer de son intégrité au moyen d'une signature cryptographique. ■

Il est possible également de lancer l'exécution du programme C dans un processus séparé, en communiquant par messages. Cela permet de s'appuyer sur le cloisonnement entre processus fourni par le système d'exploitation.

Fonctions `unsafe_*`

Les fonctions `unsafe_get` et `unsafe_set` (des modules mentionnés dans l'introduction de la section) sont des opérations d'accès et de modification dans les tableaux et chaînes de caractères, qui sont effectuées sans vérification de débordement. Elles peuvent être appelées directement ou à la place de `get` et `set` si l'option `-unsafe` du compilateur est choisie. Ces fonctions peuvent permettre d'accéder aux données manipulées par le programme en cours d'exécution.

Danger D-12: Invalidation des vérifications de débordement par les fonctions `unsafe_*` *et l'option* `-unsafe`*.*

Recommandation R-16: Proscrire l'usage des fonctions `unsafe_*` *et de l'option* `-unsafe`*.*

L'exemple suivant illustre ce danger en considérant un programme principal (`programme.ml`) manipulant une donnée confidentielle (ici un mot de passe en lignes 3–9). Par la suite, le programme fait appel à une bibliothèque externe pour effectuer une action (ligne 11). Si cette bibliothèque externe était malveillante, elle pourrait utiliser une fonction de manipulation de chaînes de caractères sans vérification de bornes (ligne 8) pour pouvoir chercher dans la mémoire l'occurrence de la chaîne `Password:` et ainsi retrouver le mot de passe manipulé par la première partie du programme.

```
1 (* programme.ml *)
2
3 print_string "Password ? ";; let p = read_line ();;
4 Password ? abcdef
5 val p : string = "abcdef"
6 let secret = Printf.sprintf "%s %8s" "password is" p;;
7 val secret : string = "password is abcdef"
8 Printf.printf "Secret: %s\n" secret;;
```

```
9  Secret: password is    abcdef
10
11 Module_malveillant.action ();;
12 Secret vole:    abcdef
```

```
1  (* module_malveillant.ml *)
2
3  let action () =
4    let base = String.make 1 'c' in
5    let buf = String.make 20 'x' in
6    let ofs = ref 0 in
7    begin try while true do
8      String.unsafe_blit base !ofs buf 0 20;
9      if buf.[0] = 'p' && String.sub buf 1 10 = ↵
      "assword:" then raise Exit;
10     incr ofs;
11   done with Exit -> () end;
12   Printf.eprintf "Secret vole: %s\n" (String.sub buf ↵
      12 8);
13 ;;
14 val action : unit -> unit
```

Dans cet exemple, l'appel à la bibliothèque externe est fait statiquement mais pourrait être fait dynamiquement par chargement dynamique du module malveillant en remplaçant la ligne 11 par la phrase :

`Dynlink.loadfile "module_malveillant.cmo";;`

et en ajoutant l'appel à la fonction `action` en fin du module malveillant (phrase `action ();;`).

Fonction `String.create`

L'étude de la fonction `create` du module `String` est faite dans la section 1.5.1.

Module `Printexc`

Les problèmes relatifs à l'utilisation des fonctions du module `Printexc` et des exceptions sont discutés dans [ANA-SECU, 2011].

Égalité, comparaison et hachage

Les problèmes relatifs au contournement de l'encapsulation par l'égalité, la comparaison et le hachage sont discutés dans [ANA-SECU, 2011]. Ce sujet est également abordé du point de vue de la représentation interne en section 1.2.4.

Ce problème est localisé aux utilisations de valeurs sensibles encapsulées. Il est donc nécessaire d'en avoir conscience lors d'importations de bibliothèques ayant accès à ces valeurs encapsulées.

Récapitulatif

Le contournement du typage et de l'encapsulation peut avoir plusieurs conséquences de natures différentes.

– Lecture complète de la mémoire : par exemple, la fonction `unsafe_get` permet de lire le contenu de la zone mémoire du processus,
– Modification de la mémoire : la fonction `unsafe_set` permet potentiellement de modifier la valeur d'une chaîne de caractères hors de ses limites et donc de modifier n'importe quelle zone mémoire du processus sauf celles en lecture seule (les conséquences diffèrent entre les modèles d'exécution, voir la section 1.5.1),
– Modification des pointeurs de code : il est possible avec `unsafe_set` de modifier le pointeur de code d'une fermeture pour qu'il pointe vers n'importe quel jeu d'instructions dans une zone mémoire arbitraire, y compris hors de la zone de code du programme, ceci permet donc de créer dynamiquement du code (ici encore voir la section 1.5.1),
– Modification de la représentation interne des valeurs (y compris non mutables) : par exemple un enregistrement à champs non-mutables peut être modifié en lui attribuant avec `Obj` un type enregistrement avec champs mutables,
– Contournement de l'encapsulation : ceci permet la lecture partielle (par exemple, en utilisant le hachage ou `Marshal`) ou complète (en changeant par exemple le type d'un type abstrait en un type explicite) de valeurs.

Danger D-13: Possibilité de contournement de l'analyse statique par utilisation de constructions non-sûres.

Recommandation R-17: Éviter l'emploi de constructions non-sûres.
Toutes les constructions non-sûres sont à proscrire absolument,

sauf le mot-clé `external` s'il est indispensable d'utiliser des fonctions externes (par exemple écrites en C), et le module `Marshal` s'il faut transmettre ou recevoir des données d'un autre programme OCaml qui utilise le format de `Marshal`. ■

Recommandation R-18: Bien choisir les options de compilation pour préserver les garanties apportées par l'analyse statique.
Certaines vérifications du compilateur ne produisent que des *warnings* et non des erreurs. L'option `-warn-error` avec l'argument `+a` permet de stopper la compilation en cas de présence d'un *warning* quelconque (l'option `-w` avec l'argument `+a-4` permet d'afficher tous les *warnings* mis à part celui indiquant la présence de filtrages fragiles).
L'option `-unsafe` désactive un certain nombre de vérifications, il est donc important de ne pas l'utiliser. ■

1.2.6 Compilation séparée, fichiers objet `.cmi`

OCaml propose un mécanisme de compilation séparée permettant de compiler chaque fichier séparément et de lier ces fichiers à la compilation de l'exécutable.

Chaque fichier source d'implémentation (extension `.ml`) est considéré comme un module. Il est composé d'une succession de phrases *toplevel* (déclarations, définitions ou expressions à évaluer). Le code compilé des déclarations (`let` et `let rec`) produit à l'exécution des valeurs dans le tas, suivant les représentations décrites en section 1.2.3. Les modules sont représentés par un enregistrement de valeurs et de fonctions.

Il n'existe pas de notion de fonction *main* en OCaml. Chaque fichier compilé a un **point d'entrée** qui exécute les phrases du texte source et crée l'enregistrement des champs du module. Un exécutable regroupe les codes compilés des modules le composant et appelle chaque point d'entrée.

Un texte source d'extension `.mli` est appelé **fichier d'interface**. Il est compilé en un **fichier objet d'interface** d'extension `.cmi`. La compilation d'un fichier source d'implémentation `.ml` donne aussi lieu à la production d'un fichier `.cmi` s'il n'existe pas de fichier d'interface `.mli` correspondant.

Un fichier `.cmi` contient le nom du module correspondant à l'unité de compilation, l'interface compilée de ce module, le hachage MD5 de l'interface compilée et la liste des noms des unités de compilation requises avec le hachage MD5 de

leurs interfaces compilées.

Il est à noter que le hachage MD5 se fait à partir des informations significatives du fichier `.mli`. La présence de commentaires ou d'une indentation particulière ne changera pas le hachage MD5 d'une interface.

La liste par défaut des unités requises contient le nom du fichier lui-même et l'unité `Pervasives`. S'y ajoutent les noms des unités importées par le fichier. De plus, en cas d'utilisation de traits spécifiques du langage, les unités correspondantes sont ajoutées automatiquement par OCaml :

- les unités `Int32`, `CamlinternalOO` et `Obj` pour les traits objet,
- les unités `Int32`, `CamlinternalMod` et `Obj` pour les modules récursifs,
- l'unité `CamlinternalLazy` pour le filtrage de valeurs paresseuses,
- les unités `BigArray`, `Unix` et `Complex` pour les accès aux champs de grands tableaux (`x.{1}`),
- l'unité `String` pour les accès aux champs de chaînes de caractères (`x.[1]`),
- l'unité `Array` pour les accès aux champs de tableaux (`x.(1)`).

> *Intérêt I-8: Compilation séparée assurée par le hachage MD5 des interfaces.*
> Les hachages MD5 des interfaces assurent la cohérence de la compilation séparée. Elles garantissent le respect des dépendances inter-fichiers. ■

La compilation de chaque fichier d'implémentation requiert la compilation au préalable des interfaces des unités qu'il importe.

Les autres fichiers objet OCaml sont spécifiques à un mode de compilation. Voir la section 2.1.3 pour la compilation séparée native et les fichiers objet `.cmx`, `.cmxs` et `.cmxa`, et la section 3.1.3 pour la compilation séparée bytecode et les fichiers objet `.cmo` et `.cma`.

Un utilitaire, `ocamlobjinfo`, permet d'afficher les informations des fichiers objet OCaml.

1.2.7 Options de compilation

Cette section décrit les options de compilation, communes aux modèles natif et bytecode, qui peuvent avoir une incidence sur la sécurité. Les options spécifiques à chacun des modèles sont décrites dans la section 2.1.1 pour le modèle natif, dans la section 3.1.1 pour le modèle bytecode, et dans le chapitre 4 pour la boucle interactive. La liste complète des options des différentes commandes

de compilation OCaml et leurs recommandations d'utilisation associées sont récapitulées dans la table 5.2. Les options de la machine virtuelle sont décrites séparément dans la section 3.3.1.

`-annot` Enregistre dans des fichiers **`.annot`** les informations de type, de portée des identificateurs, du caractère terminal ou non des appels et de localisation de chaque sous-expression du code source.

> *Recommandation R-19: Utiliser l'option* **`-annot`** *de manière à pouvoir vérifier la cohérence du texte source avec la spécification.*
> Sur les données générées, on peut vérifier qu'un appel est bien terminal et visualiser la portée d'un identificateur. Les environnements de développement fournissent ces informations de manière contextuelle. ■

`-I <dir>` Indique dans quels répertoires le compilateur doit chercher les bibliothèques du programme.

> *Danger D-14: Substitution de bibliothèques par attaque des répertoires de recherche.*

> *Recommandation R-20: Contrôler les répertoires de recherche pour éviter le chargement de bibliothèques malveillantes.*

`-linkall` Impose la liaison de tous les modules d'une bibliothèque **`.cma`**/**`.cmxa`** même si ceux-ci ne sont pas référencés dans le code source du programme. Il est important de noter que cette option change la liste des modules chargés, et donc des effets de bord éventuels réalisés lors du chargement.

`-noassert` Désactive la vérification des assertions du code source [7]. Il est déconseillé d'utiliser cette option car les assertions permettent d'identifier des comportements anormaux du programme.

`-noautolink` Désactive le chargement automatique des bibliothèques en code natif (fichiers **`.o`** ou **`.so`** compilés à partir de codes C ou d'autres langages) requises par une bibliothèque **`.cmxa`**/**`.cma`** (bibliothèques indiquées par les options **`-cclib`** et **`-ccopt`** lors de la compilation de la bibliothèque).

7. Il est à noter que la désactivation des assertions et l'activation du mode débogue sont indépendantes.

> *Recommandation R-21: Indiquer explicitement les bibliothèques à charger avec* `-cclib` *et interdire tout chargement automatique avec* `-noautolink`*.*
> Il est nécessaire de maîtriser complètement le chargement de bibliothèques et de vérifier le processus de compilation de ces bibliothèques. ■

`-nostdlib` Désactive le placement du répertoire de la bibliothèque standard dans les chemins de recherche spécifiés par l'option `-I`.

> *Recommandation R-22: Fournir explicitement les répertoires des bibliothèques utilisées, y compris celui de la bibliothèque standard.*
> Il est conseillé de vérifier l'intégrité des bibliothèques utilisées (droits d'accès, origine et modifications). ■

`-pp` Lance une phase de *preprocessing* du code source avant les autres phases de compilation. Le préprocesseur étant un programme externe, il est recommandé de contrôler son intégrité. Il est possible d'utiliser un *preprocessing* pour systématiser certaines vérifications de codes source (cf. section 1.2.1).

`-rectypes` Autorise la présence de types récursifs arbitraires (où la récursion n'est pas contrôlée par un constructeur) qui changent le comportement du typage.

> *Recommandation R-23: Ne pas utiliser l'option* `-rectypes`*.*

`-strict-sequence` Impose que chaque expression d'une séquence ait le type `unit` (sauf la dernière qui donne le résultat de la séquence). L'avertissement de compilation 10 signale aussi qu'une expression d'une séquence n'a pas le type `unit`[8].

Cette option permet de détecter des erreurs de programmation.

> *Recommandation R-24: Utiliser la vérification stricte des séquences d'expressions.*

`-unsafe` Désactive les vérifications des bornes lors des accès aux tableaux et aux chaînes. Il est recommandé de ne pas utiliser cette option car elle invalide les garanties du typage, voir la section 1.2.5 pour plus de détails.

8. L'option `-strict-sequence` et l'avertissement 10 diffèrent en cas d'expressions polymorphes : l'option `-strict-sequence` leur attribue le type `unit`.

-w <list> et -warn-error <list> L'option **-w** avec le paramètre **+a-4** active tous les avertissements de compilation mise à part celui indiquant la présence de filtrages fragiles. L'option **-warn-error +a** considère comme des erreurs de compilation tous les avertissements. La liste des avertissements du compilateur est obtenue à l'aide de l'option **-warn-help** (cf. table 1.1).

Il est recommandé d'activer ces deux options (voir la section 1.2.5 pour plus de détails).

Il est à noter que pour les futures versions du compilateur, la liste des avertissements est susceptible d'évoluer et donc de nécessiter un portage d'un programme.

	Avertissements activés par défaut
1	Suspicious-looking start-of-comment mark.
2	Suspicious-looking end-of-comment mark.
3	Deprecated syntax.
5	Partially applied function : expression whose result has function type and is ignored.
8	Partial match : missing cases in pattern-matching.
10	Expression on the left-hand side of a sequence that doesn't have type "unit" (and that is not a function, see warning number 5).
11	Redundant case in a pattern matching (unused match case).
12	Redundant sub-pattern in a pattern-matching.
13	Override of an instance variable.
14	Illegal backslash escape in a string constant.
15	Private method made public implicitly.
16	Unerasable optional argument.
17	Undeclared virtual method.
18	Non-principal type.
19	Type without principality.
20	Unused function argument.
21	Non-returning statement.
22	Camlp4 warning.
23	Useless record "with" clause.
24	Bad module name : the source file name is not a valid OCaml module name.
25	Pattern-matching with all clauses guarded. Exhaustiveness cannot be checked
26	Suspicious unused variable : unused variable that is bound with "let" or "as", and doesn't start with an underscore ("_") character.
30	Two labels or constructors of the same name are defined in two mutually recursive types.
	Avertissements désactivés par défaut
4	Fragile pattern matching : matching that will remain complete even if additional constructors are added to one of the variant types matched.
6	Label omitted in function application.
7	Some methods are overridden in the class where they are defined.
9	Missing fields in a record pattern.
27	Innocuous unused variable : unused variable that is not bound with "let" nor "as", and doesn't start with an underscore ("_") character.
28	Wildcard pattern given as argument to a constant constructor.
29	Unescaped end-of-line in a string constant (non-portable code).

TABLE 1.1 – Liste des avertissements de compilation (option `-warn-help`)

1.3 Édition de liens et chargement

Les programmes OCaml peuvent utiliser du code externe soit directement soit par appel à la bibliothèque standard. Le code externe peut être chargé :

- pendant la compilation (**chargement statique**) réalisé par le compilateur,
- au démarrage du programme (**chargement au démarrage par exécutif C**) réalisé par le système d'exploitation ou par l'exécutif de C,
- au démarrage du programme (**chargement au démarrage par la machine virtuelle**) réalisé par la machine virtuelle via le mécanisme de bibliothèques partagées du système d'exploitation,
- au cours de l'exécution à la demande du code exécuté (**chargement dynamique**) réalisé par appel au module `Dynlink`.

1.3.1 Chargement statique

L'édition de liens et le chargement statique forment la dernière étape de la compilation. Dans le compilateur natif, ils sont pour l'essentiel délégués à l'éditeur de liens du système d'exploitation : le compilateur OCaml calcule la liste des fichiers objet à inclure dans le programme et ajoute un code d'initialisation (qui appelle tour à tour les points d'entrée des modules du programme). Tous ces fichiers sont ensuite passés en argument à l'éditeur de liens du système, qui produit alors le programme exécutable.

Lorsque l'utilisateur spécifie une bibliothèque à lier avec son programme, le compilateur OCaml ne lie pas forcément tous les modules de cette bibliothèque : il ne lie que les modules dont au moins une valeur est utilisée par le programme. Ceci est vrai aussi bien pour les bibliothèques de l'utilisateur que pour la bibliothèque standard. Ce comportement peut être modifié avec l'option `-linkall` du compilateur. Si cette option est utilisée, tous les modules des bibliothèques (`.cmxa`/`.cma`) passés en arguments au compilateur seront liés et leurs points d'entrée exécutés au démarrage du programme.

> *Recommandation R-25: En cas d'utilisation de* `-linkall`*, vérifier toutes les bibliothèques liées.*
> En cas d'utilisation de l'option `-linkall`, il est recommandé de vérifier les sources de toutes les unités de compilation de toutes les bibliothèques passées en ligne de commande (sans se limiter aux unités réellement utilisées par le programme). ■

Le compilateur n'autorise la liaison que d'une seule unité de compilation d'un

nom donné : deux unités de compilation `Foo` de deux bibliothèques `a.cma` et `b.cma` ne pourront pas être liées simultanément.

1.3.2 Chargement au démarrage par exécutif C

L'exécutif OCaml, étant programmé en C, utilise l'exécutif C et des bibliothèques du langage C qui peuvent être chargées au démarrage par l'exécutif C. Ce chargement est transparent du point de vue du programmeur. Les problèmes de sécurité posés par ce mécanisme ne sont pas spécifiques à OCaml et ne sont donc pas abordés dans ce document. Ceci concerne aussi bien le modèle natif que le modèle bytecode.

Pour éviter ce chargement au démarrage par l'exécutif C, il faut charger l'exécutif OCaml avec l'option `-static` du *linker* C, ce qui peut être fait de la manière suivante :

- en utilisant l'option `-cclib -static` avec le compilateur natif (`ocamlopt`) ou avec le compilateur bytecode particularisé (`ocamlc -custom`),
- en recompilant la machine virtuelle `ocamlrun` avec l'option `-static` du compilateur C.

Il est à noter qu'avec MacOS X, il n'est pas possible d'éviter le chargement au démarrage natif à cause des particularités de ce système.

1.3.3 Chargement au démarrage par la machine virtuelle

Dans le cas du modèle bytecode, le chargement au démarrage par la machine virtuelle concerne le code externe utilisé par le programme et utilisé par la construction `external`, lorsque l'option `-custom` du compilateur n'est pas activée. Voir la section portant sur l'interfaçage avec du code C en modèle bytecode, section 3.2.3.

1.3.4 Chargement dynamique de code OCaml compilé

Quel que soit le mode d'exécution, un programme peut charger du code OCaml compilé afin de l'exécuter au cours de sa propre exécution. Le module `Dynlink` permet de charger du code OCaml natif depuis un programme OCaml natif et du bytecode depuis un programme OCaml bytecode. Toutefois, dans le modèle natif, le code doit être compilé spécifiquement avec l'option `-shared` pour pouvoir être chargé dynamiquement. OCaml ne propose pas de chargement dynamique croisé natif–bytecode ou bytecode–natif.

Ce chargement dynamique au cours de l'exécution ne peut résulter que de l'utilisation du module `Dynlink` dans le programme appelant. Le programme appelle la fonction `loadfile` avec en argument le nom du fichier contenant le code à charger.

> *Danger D-15: Risque de manipulation malveillante de l'argument de* `Dynlink.loadfile`*.*
> Cet argument est une chaîne de caractères arbitraire qui peut provenir d'une entrée utilisateur à l'exécution. ■

> *Recommandation R-26: Contrôler la provenance ou valider l'argument de* `Dynlink.loadfile`*.*

> *Danger D-16: Modification malveillante du fichier chargé.*
> Le fichier à charger peut être modifié en substituant la version compilée d'un code OCaml malveillant. ■

> *Recommandation R-27: Contrôler les droits d'accès aux fichiers susceptibles d'être chargés dynamiquement.*

Le chargement effectue l'édition de liens pour le code chargé, ce qui met à jour la table d'indirection des pointeurs de code du code chargé, en utilisant les informations de la table des symboles du programme appelant. Le code chargé est ensuite aussitôt exécuté (séquence d'initialisation), ce qui conduit potentiellement à des effets de bord et des créations de valeurs dans le tas.

Le code appelant ne peut pas accéder aux fonctions du code chargé, sauf si (1) celui-ci met à jour des références de fonctions prévues à cet effet dans le code appelant ou si (2) le code appelé appelle une fonction du code appelant en lui passant en argument l'une de ses propres fonctions. Le code appelé peut accéder aux modules du code appelant, à condition que les interfaces de ces modules aient été fournies au code appelé au moment de sa propre compilation. Le compilateur inclut, dans les codes compilés de l'appelant et de l'appelé, les hachages MD5 des interfaces compilées, importées et exportées par l'appelant et l'appelé. Il utilise ces hachages MD5 pour vérifier la cohérence de l'édition de liens.

> *Intérêt I-9: La vérification des hachages MD5 des interfaces garantit la cohérence du typage.*

Cette vérification évite les problèmes de changement de version, qui pourraient conduire à des incohérences de typage. ■

Danger D-17: La vérification des hachages MD5 ne protège pas contre une modification délibérée du fichier chargé.

L'utilisation des hachages MD5 est la seule vérification dynamique apportant des garanties sur la cohérence du typage. Cette vérification permet d'éviter des erreurs de programmation mais pas des malveillances. Son contournement peut conduire à un contournement du typage et de l'encapsulation (cf. section 1.2.5).

Danger D-18: Risque de contournement du typage et de l'encapsulation par utilisation du module `Dynlink`.

Recommandation R-28: Éviter les utilisations de `Dynlink`.
Si le chargement dynamique de code est absolument nécessaire, il est recommandé de faire la lecture critique des textes source des codes à charger, de les compiler dans un environnement de confiance et de vérifier les droits d'accès aux fichiers objet correspondants. ■

Configuration globale de `Dynlink`

Le module `Dynlink` permet de contrôler à quels modules le code chargé peut avoir accès. La fonction `prohibit` précise la liste de noms des modules interdits et la fonction `allow_only` restreint les modules référençables à ceux d'une liste donnée. Ces fonctions sont à appeler depuis le programme contenant l'appel à `Dynlink`, chacun de leurs appels restreignant la liste des modules référençables. On peut donc empêcher l'accès par le code chargé à des modules `Obj` ou `Marshal`. En bytecode ce contrôle peut être étendu aux `external` avec la fonction `allow_unsafe_modules`. Cependant, ce contrôle porte sur des modules complets, ce qui peut être difficile à adapter aux besoins : pour proscrire la fonction `unsafe_get` du module `String`, il faut interdire `String` entièrement. De plus, ce contrôle s'appuie sur le mécanisme d'interfaces et requiert donc un contrôle de la chaîne de compilation du fichier chargé (voir danger D-2 et recommandation R-4).

Recommandation R-29: Ne pas se reposer entièrement sur le contrôle d'accès fourni par `Dynlink`.

Si l'utilisation de `Dynlink` est requise par la spécification du programme, il est recommandé de contrôler la liste des modules référençables. ■

Dans l'exemple suivant, le texte source du programme appelant interdit l'appel au module `String` ainsi qu'aux fonctions externes.

```
Dynlink.prohibit ["String"];;
Dynlink.allow_unsafe_modules false;;
Dynlink.loadfile "a.cmo";;
```

Le chargement du module `A` suivant lancera une exception `Unavailable_unit`.

```
(* a.ml *)
...  String.unsafe_get ...
external ...
...
```

Une fonction `reset` permet de réinitialiser la configuration du module `Dynlink`. Des fonctions obsolètes (marquées *deprecated* mais maintenues pour des raisons de compatibilité ascendante) `add_interfaces` et `add_available_units` ajoutent des noms de module dans la liste des modules référençables. La configuration de `Dynlink` étant globale, les fonctions de configuration permettent de modifier la configuration antérieure. Les fonctions `reset`, `add_interfaces` et `add_available_units` rendent la nouvelle configuration plus permissive. Il est donc conseillé de les proscrire.

Recommandation R-30: Proscrire les utilisations des fonctions `reset`, `add_interfaces` *et* `add_available_units` *du module* `Dynlink`.

Pour contrôler plus finement les utilisations de `Dynlink`, il est préférable de créer un module ne contenant que des fonctions sûres et de n'autoriser que ce module pour les modules chargés dynamiquement en combinant cette restriction avec `allow_unsafe_modules false`. Ce contrôle approfondi n'est possible qu'en bytecode car en natif `allow_unsafe_modules` n'a aucun effet.

Danger D-19: Les appels externes d'un code natif chargé dynamiquement ne sont pas contrôlés par `allow_unsafe_modules`.

Le texte source suivant donne un exemple d'un module ne contenant que des fonctions sûres qui peut être utilisé avec la configuration `allow_only ["ModuleSur"]` et `allow_unsafe_modules false` pour avoir un contrôle fin du

chargement. Ce contrôle doit s'acompagner de la maîtrise de la compilation du fichier chargé dynamiquement.

```
1 (* moduleSur.ml *)
2 let string_set = String.set
3 let string_get = String.get
4 (* on n'inclut ni String.unsafe_set ni ↵
    String.unsafe_get *)
```

Recommandation R-31: Contrôler le chargement dynamique.
Pour renforcer la sécurité du chargement dynamique, il est nécessaire de compiler les fichiers chargeables dynamiquement en contrôlant les modules auxquels ils font référence et, pour le chargement dynamique de bytecode, l'absence d'appels externes comme détaillé ci-dessus. ■

1.3.5 Interfaçage avec du code en C

Du code natif, écrit le plus souvent en C, peut être appelé depuis un texte source OCaml à l'aide de la construction `external`. Ces appels sont typés par le compilateur OCaml, selon les déclarations jointes à la construction `external` par le programmeur. Rien ne peut assurer automatiquement que les fonctions C ainsi appelées respectent le type indiqué. Ce point est détaillé dans la partie décrivant `external` en section 1.2.5.

L'interfaçage avec C est utilisé également pour les primitives du langage et de la bibliothèque standard, les appels système, les grosses bibliothèques externes (calcul matriciel flottant, interface graphique, etc). Là encore, le compilateur ne peut pas détecter les incohérences entre le type annoncé dans le code OCaml et le type des fonctions C. On peut cependant accorder une certaine crédibilité aux utilisations de `external` dans la bibliothèque standard, au vu des utilisations et relectures qui en sont faites.

1.4 Parties communes des exécutifs

L'**exécutif** ou système *runtime* est constitué des parties de code communes à tous les exécutables OCaml. Il est ajouté par le compilateur et inclut la gestion mémoire et le GC, la gestion des exceptions, l'interfaçage avec le système d'exploitation et les bibliothèques standards écrites en C (comparaison polymorphe,

marshalling...). Il existe deux versions de l'exécutif : une pour le modèle natif et une pour le modèle par machine virtuelle (qui sert aussi pour le modèle interactif). Ces deux versions de l'exécutif partagent une grande partie de leur code. Elles sont écrites en C et en assembleur et représentent une couche intermédiaire entre le système d'exploitation et le programme OCaml.

Les exécutifs des différents modèles d'exécution partagent :

- la gestion mémoire (traitée en section 1.5),
- la gestion des signaux (traitée ci-dessous, voir 1.4.1),
- la couche bas niveau de la bibliothèque standard :
  - la gestion des entrées-sorties,
  - les fonctions sur les chaînes de caractères (dont le non-débordement et le débordement des bornes sont traités en sections 1.2.2 et 1.2.5),
  - les primitives du module `Printexc` d'impression des exceptions (traitées dans [ANA-SECU, 2011]),
  - les interpréteurs des tables produites par `Yacc` et `Lex`,
  - les primitives du type `obj` (traitées en section 1.2.5),
  - les fonctions de sérialisation/désérialisation (traitées en section 1.2.5),
  - les fonctions de hachage (traitées en section 1.2.4),
  - les fonctions sur les entiers et les flottants,
  - les primitives de comparaison polymorphes (traitées en section 1.2.4), et
  - les primitives sur les tableaux (dont le non-débordement et le débordement des bornes sont traités en sections 1.2.2 et 1.2.5).

Les parties des exécutifs qui diffèrent selon le modèle sont :

- l'interpréteur de bytecode,
- le mode débogue d'exécution spécifique au bytecode (cf. section 3.3.3),
- la gestion des exceptions,
- le chargeur dynamique de modules `Dynlink` (traité dans les sections 1.3.4, 2.2.2 et 3.2.2),
- une partie du GC (ajustement des appels),
- le module `Callback`,
- une partie de la gestion des signaux spécifique au modèle natif.

Il existe quelques différences entre l'exécutif chargé avec le code natif généré et l'exécutif de la machine virtuelle pour l'exécution de bytecode. La différence principale étant que l'exécutif de la machine virtuelle contient un interpréteur de bytecode. Les autres différences mineures sont détaillées en section 5.3.

1.4.1 Traitement des signaux

Le gestionnaire des signaux fait partie de l'exécutif OCaml. La définition et la réaction aux signaux est commune aux modèles natif et bytecode. La vérification de la présence d'un signal diffère : elle se fait à chaque allocation en natif (voir section 2.3.1) et à chaque instruction en bytecode (voir section 3.3.2). Quel que soit le modèle, la synchronisation du gestionnaire des signaux ne peut pas intervenir à un moment où le GC opère, garantissant ainsi l'initialisation des pointeurs.

> *Intérêt I-10: Garantie d'initialisation des pointeurs par non interférence entre traitement des signaux et GC.*

Le gestionnaire de signaux se configure avec les fonctions `signal`, `set_signal` et `catch_break` du module `Sys` et la fonction `sigprocmask` du module `Unix`. Ces fonctions permettent d'associer un traitement à un signal donné ou de masquer un signal, en positionnant des variables globales. Ces variables sont nécessairement mutables et il n'est pas possible d'interdire d'éventuelles affectations ultérieures.

> *Danger D-20: Re-configuration du gestionnaire de signaux par un texte source non-digne de confiance.*

> *Recommandation R-32: Contrôler les configurations du gestionnaire de signaux.*

> *Recommandation R-33: Exclure les modules `Sys` et `Unix` en cas de chargement dynamique.*
> En cas de chargement dynamique, il est conseillé d'exclure les modules `Sys` et `Unix` en utilisant la fonction `prohibit` du module `Dynlink` (cf. section 1.3.4) pour empêcher la re-configuration du gestionnaire de signaux. ■

La gestion des signaux par OCaml suit la sémantique des signaux Unix :

- Lors du rattrapage d'un signal, ce signal est masqué pendant l'exécution de la fonction associée. Celle-ci ne peut donc pas être interrompue par le même signal, par contre l'émission d'un autre signal peut l'interrompre.
- Lorsqu'un signal masqué est reçu, sa présence est enregistrée et sa fonction associée sera exécutée dès le démasquage du signal.

– Si un signal est reçu plusieurs fois avant d'être traité, la fonction associée ne sera exécutée qu'une seule fois.

1.4.2 Gestion des *Threads*

OCaml propose un mécanisme de gestion des *Threads* (processus légers partageant la mémoire). Deux implémentations sont disponibles suivant le système d'exploitation :

- *Threads* au niveau système. Cette implémentation s'appuie directement sur les *Threads* fournis par le système d'exploitation (*POSIX Threads* sous Unix et *Win32 Threads* sous Windows). Cette implémentation n'est pas disponible sur tous les systèmes mais lorsqu'elle l'est, elle gère les modèles natif et bytecode. Pour utiliser cette implémentation, il faut compiler le programme OCaml avec l'option `-thread`.
- *Threads* au niveau de la machine virtuelle. Cette implémentation est uniquement disponible en modèle bytecode sous Unix. Elle implémente un système de temps partagé dans la machine virtuelle. Pour utiliser cette implémentation, il faut compiler le programme OCaml avec l'option `-vmthread`.

Dans les deux cas, le modèle mémoire est hérité de la plateforme sous-jacente.

1.5 Gestion automatique de la mémoire

La gestion de la mémoire est une partie commune aux exécutifs des différents modèles d'exécution. Il est important de noter que cette gestion de la mémoire est automatique. Elle est effectuée par le compilateur et l'exécutif de OCaml. Le compilateur détermine lors du typage l'espace mémoire requis pour chaque valeur et la portée de chaque identificateur. L'exécutif organise la mémoire et ses accès (section 1.5.1) et le ramasse-miettes (GC) gère dynamiquement cette mémoire (section 1.5.2).

> *Intérêt I-11: Automatisation de la gestion mémoire.*
> La gestion de la mémoire est automatique et n'est plus à la charge du programmeur évitant ainsi les dangers inhérents à la manipulation de pointeurs. ■

1.5.1 Allocation, désallocation

La compilateur utilise les informations de typage pour déterminer à la compilation la taille et le format des zones de mémoire à allouer aux données, conformément à la description faite dans la section 1.2.3. Seules les informations sur la taille sont préservées dans le code compilé (code natif ou bytecode). Elles sont utilisées par les fonctions d'allocation qui décident, à l'exécution, du placement des zones mémoire correspondantes.

Il n'existe pas de primitive d'allocation équivalente au `malloc` de C, ni de manipulation explicite de pointeurs. Il n'y a pas de variables non ou partiellement initialisées car la déclaration d'une variable, l'allocation de son espace mémoire, et son initialisation globale se font simultanément.

> *Intérêt I-12: Absence d'erreurs d'allocation mémoire.*
> La gestion automatique de l'allocation mémoire et le fait que déclaration, allocation et initialisation sont indissociables en OCaml, évitent un grand nombre d'erreurs de programmation permettant des attaques. ■

Les accès en mémoire sont contrôlés par l'obligation de respecter le typage.

> *Intérêt I-13: Absence d'erreurs des accès en mémoire.*
> Dans un programme ne contournant pas le typage (voir la section 1.2.5), la gestion de la mémoire est garantie correcte. ■

Organisation de la mémoire

Dans les systèmes d'exploitation classiques, notamment sur ceux pour lesquels OCaml est disponible, la mémoire d'un processus est divisée en plusieurs zones qui sont utilisées par diverses parties du programme selon leurs besoins :

- La **zone de code** est restreinte en lecture seule par le système d'exploitation. Elle contient le code natif exécutable ainsi que les constantes déclarées dans le programme.
- La **pile** est utilisée pour stocker les arguments passés aux fonctions et les variables locales.
- Le **tas** constitue généralement la plus grosse zone, et sert à stocker les valeurs calculées par le programme. Dans un processus OCaml, le tas est subdivisé en tas C et tas Caml (voir section 1.5.2).

Dans le modèle d'exécution natif, les données placées dans la pile et le tas par OCaml ne contiennent pas de code exécutable. Le système d'exploitation

peut donc interdire toute exécution de ces zones sans poser de problèmes à OCaml. Ceci permet d'apporter une protection supplémentaire même en cas de contournement du typage.

> *Intérêt I-14: Séparation programme–données reflétée au bas niveau dans le modèle natif.*

Dans le modèle d'exécution bytecode, seul le code de l'exécutif se trouve dans la zone de code, le bytecode lui-même et les constantes du programme sont chargés dans le tas, et ne sont donc pas en lecture seule. De plus la machine virtuelle peut interpréter n'importe quelle zone de mémoire lisible quelles que soient ses protections.

Allocation

Les allocations de mémoire dans la zone de code se font statiquement, c'est-à-dire à la compilation, alors que les allocations de mémoire dans la pile ou le tas se font dynamiquement, c'est-à-dire au cours de l'exécution. L'allocation statique permet d'effectuer à la compilation un traitement qui consommerait du temps de calcul à l'exécution. C'est au compilateur d'optimiser l'exécution d'un programme en effectuant le plus possible d'allocations statiques. Mais seules les valeurs identiques dans toutes les exécutions du programme peuvent être allouées statiquement. Le compilateur OCaml alloue statiquement les valeurs dont la constance est détectée syntaxiquement.

Par exemple (figure 1.3), pour compiler la phrase `let x = [1; 2; 3];;` le compilateur alloue statiquement la liste constante `[1; 2; 3]` dans la zone de code, et ajoute un pointeur vers cette liste dans le code compilé. Ce pointeur donne accès à la variable globale `x`. En revanche le code compilé pour la phrase `let y = 5 :: x in e;;` alloue dynamiquement la valeur `5 :: x` et place dans la pile un pointeur vers cette valeur, qui sera utilisé pour évaluer l'expression `e`.

Le mécanisme de gestion du tas est fondé sur l'allocation dynamique de la mémoire. Comme mentionné dans la section 1.2.3, la représentation ne différencie pas valeurs mutables et non-mutables : les garanties de non-modification des valeurs non-mutables sont données statiquement par le typage du code source, et ne sont plus vérifiées à l'exécution. La seule exception est en mode natif pour les valeurs constantes du programme (statiquement évaluables), qui sont placées dans la zone de code et donc protégées contre l'écriture par le système d'exploitation.

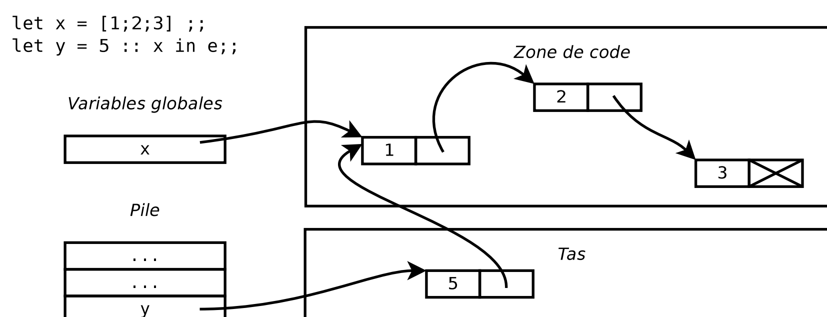


FIGURE 1.3 – Exemple d'allocation

À l'exception des valeurs scalaires (entiers, caractères, constructeurs constants) qui sont représentées par des entiers, toutes les valeurs calculées au cours de l'exécution sont représentées par des pointeurs vers des zones de mémoire allouées dynamiquement (cf. section 1.2.3). Pour les constructeurs avec arguments, la simple application du constructeur à ses arguments alloue une zone de mémoire qui représente la valeur correspondante. La construction d'un enregistrement avec la syntaxe `{label = valeur; ...}` alloue aussi une zone de mémoire qui représente le nouvel enregistrement. Notons que certains types disposent d'une syntaxe spéciale, `[ ; ; ]` pour les listes, `( , , )` pour les n-uplets, et `[| ; ; |]` pour les tableaux, qui cache l'application d'un constructeur et qui alloue donc une zone mémoire pour la valeur ainsi décrite. Pour les tableaux et les chaînes, l'allocation peut aussi être effectuée par l'appel des fonctions `Array.make`, `Array.create` et `String.make` qui initialisent la zone allouée. En revanche, la fonction `String.create` effectue une allocation mémoire sans initialisation (pour des raisons de performance). L'appel à la fonction `String.create` peut donc permettre de consulter des portions de mémoire précédemment allouées sans avoir à contourner le typage.

> *Danger D-21: L'allocation des chaînes de caractères peut permettre une consultation illicite de la mémoire.*
> Si le texte source appelle la fonction `String.create` juste après la désallocation de données sensibles, celles-ci risquent d'être révélées dans la chaîne créée. ■

> *Recommandation R-34: Interdire l'utilisation de* `String.create`.
> Il est recommandé d'utiliser systématiquement `String.make` à la place de `String.create` et de faire cette substitution dans le code

source de toutes les bibliothèques utilisées par le programme. ■

Partage

Lorsque des valeurs structurées ont en commun des sous-structures, l'allocation dynamique conserve le partage exprimé dans le programme. Dans l'exemple suivant (voir figure 1.4), toute les occurrences de `a` partagent leur même valeur `[1;2;3]` car `a` est une variable. Par contre, les deux occurrences de `C (a,a)` ne sont pas partagées car ce sont des applications de constructeurs. Toute application d'un constructeur à des arguments conduit à l'allocation d'une nouvelle valeur.

```
let a = [1; 2; 3];;
let x = C (a,a);;
let y = C (a,a);;
```

Il est à noter qu'un constructeur est un identifiant commençant par une majuscule et que les constructions suivantes se comportent comme des constructeurs : `x::y`, `[x;y]`, `(x,y)`, `[|x;y|]`, `{ m=x; n=y }`, `new x`, `object method m=x end` ainsi que `fun x -> y` qui alloue une fermeture.

Dans cet exemple, la distinction n'est pas très importante car les éléments de la liste ne sont pas mutables, donc le programme fonctionnera de la même façon, qu'il y ait copie ou partage. Le partage peut par contre avoir des conséquences importantes dès que des valeurs mutables sont utilisées dans des structures de données. La gestion des structures de données mutables exige du programmeur qu'il maîtrise exactement les cas de duplication et de partage. En particulier, le partage par inadvertance d'une donnée mutable peut introduire des erreurs subtiles : c'est le problème de l'*aliasing*[9].

> *Danger D-22: Le partage d'une valeur mutable peut aboutir à des modifications non voulues de cette valeur.*
> Le danger du partage de valeurs mutables est inhérent à tout langage ayant des traits impératifs et n'est donc pas spécifique à OCaml. ■

La construction `"abcde"` est appelée une chaîne de caractères littérale et ne se comporte pas comme un constructeur. Elle n'alloue pas de nouvelle valeur bien

9. L'opérateur d'égalité physique `==` permet au programme d'observer le partage de ses données, qu'elles soient mutables ou non.

que les chaînes de caractères soient mutables. Cela peut donner des résultats surprenant comme illustré par l'exemple suivant.

```
let f () = "abcde";;
val f : unit -> string = <fun>
let s = f ();;
val s : string = "abcde"
s.[3] <- 'X';;
- : unit = ()
let t = f ();;
val t : string = "abcXe"
```

Danger D-23: Mutabilité et partage de chaînes de caractères littérales.

Recommandation R-35: Forcer la copie des chaînes de caractères littérales.
Pour éviter le partage intempestif des chaînes de caractères littérales, il est recommandé de toujours leur appliquer la fonction `String.copy`. ■

Recommandation R-36: Vérifier que le partage ne porte que sur des valeurs n'ayant aucune sous-structure mutable.
Le partage de valeurs mutables doit être soigneusement justifié et contrôlé. Il est aussi conseillé de réduire l'usage des mutables, voir [ANA-SECU, 2011]. ■

Désallocation

OCaml possède un mécanisme, le ramasse-miettes (voir section 1.5.2), pour désallouer automatiquement les zones de mémoire qui ne sont plus utilisées par le programme.

1.5.2 Ramasse-miettes (GC)

Le **ramasse-miettes** ou Garbage Collector, dénoté communément par le sigle **GC**, réalise automatiquement la désallocation des zones de mémoire qui ne sont plus utilisées par le programme. La désallocation est complètement gérée

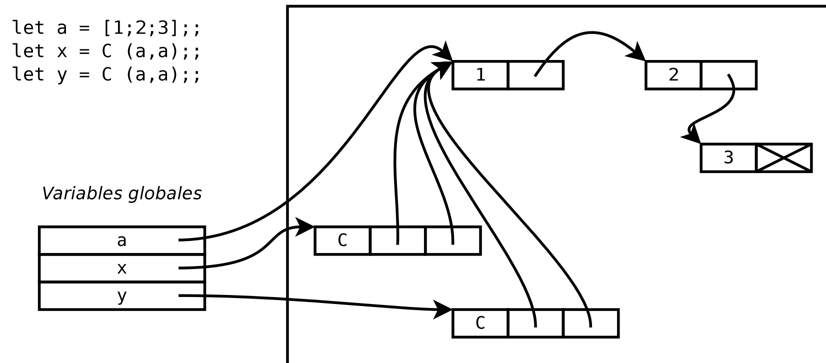


Figure 1.4 – Exemple de partage

par le GC. Aucune fonction n'est donc fournie au programmeur pour désallouer une zone de mémoire donnée. OCaml n'offre aucun équivalent à la fonction `free` de C.

> *Intérêt I-15: Absence d'erreurs de désallocation mémoire.*
> Le GC de OCaml décharge le programmeur de la gestion de la désallocation de mémoire et optimise cette gestion. Il évite ainsi les erreurs de programmation inhérentes à la gestion manuelle de la mémoire (accès à une zone mémoire libérée, libération d'une zone mémoire déjà libérée, fuite de mémoire par le maintien de zones de mémoire qui ne sont plus utilisées). ■

Structure du tas Caml

Dans la mémoire du processus (voir section 1.5.1), le sous-ensemble de la mémoire qui est alloué par les mécanismes décrits ci-dessus et désalloué par le GC s'appelle le **tas Caml**. Le tas Caml est divisé en deux parties : le **tas mineur** et le **tas majeur**. La plupart des allocations se font dans le tas mineur. Les seules exceptions sont les blocs de grande taille.

Le **tas C** est géré par le code C qui s'exécute sur la machine en même temps que le code OCaml (pour plus de détails voir les sections 2.2.3 et 3.2.3). Dans le tas C, la mémoire est allouée et désallouée explicitement avec les fonctions `malloc` et `free`. Le tas Caml est inclus dans le tas C.

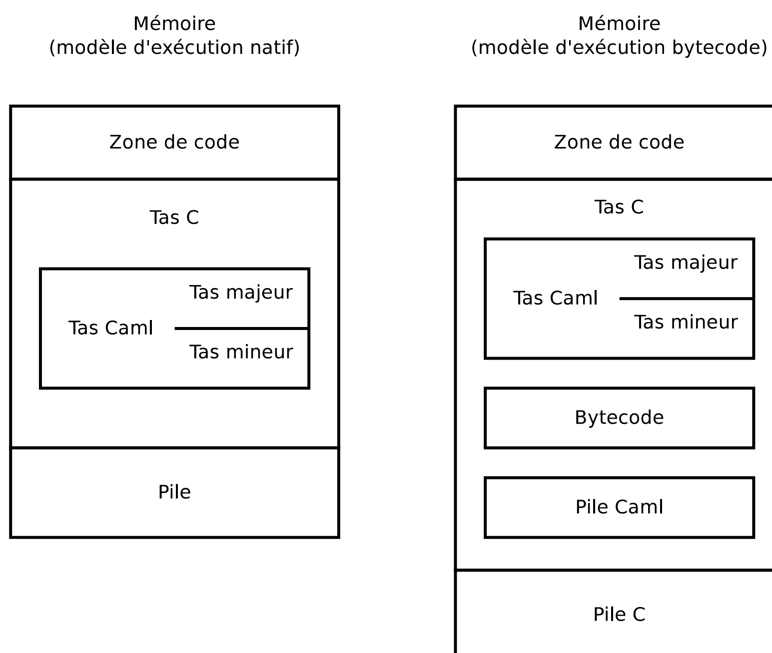


FIGURE 1.5 – Structure de la mémoire

Fonctionnement du GC

Lorsque le tas mineur est plein, le gestionnaire de mémoire arrête le programme OCaml et déclenche un **GC mineur**, qui est un algorithme dit *stop-and-copy*. Celui-ci copie les données des zones encore utiles du tas mineur vers le tas majeur, change tous les pointeurs vers ces données pour qu'ils référencent la nouvelle copie, et prépare le tas mineur pour une nouvelle phase d'allocation. Notons que les zones devenues inutiles dans le tas mineur ne sont pas explicitement réinitialisées : elles seront réutilisées par cette nouvelle phase d'allocation.

> *Danger D-24: Duplication en mémoire des données confidentielles.* Le programme ne contrôle pas le mécanisme de copie du GC. Celui-ci peut dupliquer des zones mémoire contenant des données confidentielles. Cela peut faciliter les attaques par lecture de la mémoire (voir section 1.5.3 pour une discussion approfondie). ■

> *Danger D-25: Les données inutiles du tas mineur ne sont pas effacées.*

Le **GC majeur** désalloue les zones inutiles du tas majeur, à l'aide d'un algorithme *mark-and-sweep*. Il s'agit d'un algorithme *incrémental* : il s'exécute partiellement à chaque GC mineur, ce qui évite d'interrompre le fonctionnement du programme pour de longues périodes. En contrepartie, cela retarde la désallocation des données inutiles par le GC majeur. Cet algorithme incrémental est dérivé d'un algorithme concurrent décrit dans [Doligez, 1995, Doligez et Leroy, 1993] et [Doligez et Gonthier, 1994]. Il est prouvé qu'aucune zone utile n'est désallouée et que toutes les zones inutiles sont désallouées, après leur fin d'utilisation, dès le passage du deuxième GC majeur. En pratique le GC mineur est exécuté avec une période de quelques millisecondes.

> *Danger D-26: Désallocation retardée.*
> Les données devenues inutiles ne sont pas désallouées immédiatement par le GC. ■

Bien que cet algorithme incrémental ne fournisse pas de garanties fortes sur les temps de pause dûs au GC, il donne en pratique de bons résultats en termes de latence : les arrêts du programme dûs aux GC dépassent rarement une milliseconde. Une exception notable est dûe au problème de la fragmentation : selon la suite d'allocations effectuée par le programme, il peut arriver que les zones libres du tas soient toutes trop petites pour allouer une nouvelle zone demandée, même si la quantité de mémoire libre est suffisamment grande, parce que cette mémoire libre est fragmentée en petits intervalles séparés par des zones de mémoire utilisées par le programme. Lorsqu'il détecte ce problème, le gestionnaire de mémoire déclenche un **compactage**, qui parcourt tout le tas et recopie les objets pour rassembler les zones libres en une seule grande zone libre. Ce compactage ne peut malheureusement pas se faire incrémentalement, et il arrête donc le programme pendant une durée qui peut aller jusqu'à quelques secondes si le tas est très grand (de l'ordre du gigaoctet).

Le GC de OCaml est très efficace, particulièrement lorsque le programme manipule des structures de données complexes. Citons par exemple Jon Harrop qui, sur une expérimentation décrite dans son blog [Harrop, 2010], n'a pas pu battre en C/C++ les performances du GC de OCaml.

Traits avancés du GC

Le GC fournit plusieurs mécanismes spéciaux : valeurs finalisées, pointeurs faibles, optimisation des valeurs paresseuses. Seules les valeurs finalisées seront détaillées car les autres traits n'ont pas d'influence sur la sécurité.

Le GC fournit un mécanisme de **finalisation** : le programme peut attribuer une fonction de finalisation à toute donnée allouée dans le tas. Lorsque cette donnée est devenue inutile et est sur le point d'être désallouée, le GC appelle alors la fonction de finalisation en lui donnant cette donnée en argument.

On pourrait être tenté d'utiliser la finalisation pour effacer les données confidentielles lorsqu'elles ne sont plus utilisées, mais il faut être conscient qu'il peut s'écouler un intervalle de temps relativement long (plusieurs secondes) entre le moment où une donnée n'est plus utilisée et le moment où le GC appelle la fonction de finalisation. Le problème de la gestion des données confidentielles est discuté plus en détail dans la section 1.5.3, et la recommandation R-39 préconise de ne pas utiliser la finalisation pour l'effacement des données confidentielles.

Taille du tas

Le GC n'a pas besoin de changer la taille du tas mineur, mais lorsque le taux de remplissage du tas majeur augmente, le GC l'agrandit pour éviter une dégradation des performances du programme. Si cet agrandissement n'est pas possible, le GC lève une exception `Out_of_memory` ou arrête le programme avec une erreur fatale.

Bibliothèques externes

Un programme OCaml peut charger à la fois des bibliothèques OCaml et des bibliothèques contenant du code C. Le chargement dépend du modèle d'exécution et est décrit en détail dans les sections 2.2 et 3.2. Dans le cas du chargement de bibliothèques OCaml, les garanties du typage décrites en section 1.2.2 sont préservées par le processus de compilation et par le GC. En revanche, le GC OCaml n'est pas en charge de la gestion mémoire du code C qui est faite directement dans le tas C à l'aide des fonctions `malloc` et `free`.

Il est à noter que le tas Caml est inclus dans le tas C car le GC utilise `malloc` pour agrandir le tas, et que le code C peut donc y accéder sans restriction car les codes OCaml et C sont exécutés dans le même processus.

> *Danger D-27: Accès mémoire par code C non contrôlé.*
> Un code C malveillant a accès à tout le contenu de la mémoire (tas C), y compris au tas Caml. ■

> *Recommandation R-37: Limiter l'usage des bibliothèques C.*
> L'utilisation de bibliothèques contenant du code C est à éviter s'il

n'est pas possible de vérifier ce code. ■

S'il est indispensable d'utiliser un code écrit en C, il reste possible de le lancer dans un processus séparé et de profiter ainsi des garanties de cloisonnement apportées par le système d'exploitation. Les deux processus communiquent alors par envoi de message, et il est impossible pour l'un de lire ou d'écrire la mémoire de l'autre. Une telle solution sort du cadre de l'étude du langage OCaml.

Appel explicite au GC

Le programme peut appeler des fonctions du module `Gc` de la bibliothèque standard OCaml pour déclencher explicitement un GC mineur, un GC majeur, ou même un compactage, pour désallouer les zones de mémoire qui ne sont plus utilisées.

Intérêt I-16: Libération de la mémoire par appel explicite au GC.
Un appel explicite au GC permet au programmeur de relâcher le plus tôt possible des données volumineuses réduisant ainsi la consommation mémoire du programme. ■

1.5.3 GC et données confidentielles

Lorsqu'un programme manipule des données confidentielles, il est parfois possible de l'attaquer par lecture de son espace mémoire. Cette lecture peut se faire de l'intérieur (par exemple, avec `String.create`, voir danger D-21), ou de l'extérieur (contournement des restrictions du système d'exploitation, lecture de fichiers *core dump*, de fichiers de *swap*, de fichier d'hibernation, etc). Pour se protéger contre ces attaques, il est nécessaire de contrôler les données confidentielles en mémoire pour éviter qu'elles soient dupliquées, et de les effacer dès qu'elles ne sont plus nécessaires au fonctionnement du programme.

En OCaml, toute donnée structurée (c'est-à-dire toute donnée sauf les entiers, les caractères et les constructeurs constants) est représentée et manipulée au travers d'un pointeur vers la zone de mémoire qui représente cette donnée. Le passage d'arguments aux fonctions se fait en copiant ces pointeurs dans la pile (passage d'arguments par référence), ce qui évite la duplication des données structurées dans la mémoire du processus.

Intérêt I-17: Non-duplication des arguments des fonctions.
Les données complexes ne sont jamais dupliquées lors des appels de fonction. ■

Le GC de OCaml peut dupliquer des données en mémoire dans deux cas :

- Lors d'un GC mineur, il recopie les données vers le tas majeur. Les allocations ultérieures dans le tas mineur effacent progressivement les anciennes données, mais il n'y a pas de garantie sur leur temps de survie.
- Lors d'un compactage, il recopie toutes les données du tas pour les déplacer. Certaines de ces données sont immédiatement effacées car leur espace mémoire est utilisé pour recopier d'autres données, mais là encore, le GC ne donne aucune garantie et son comportement n'est en pratique pas prévisible.

> *Danger D-28: Duplication de données par le GC.*
> Le GC peut dupliquer des données confidentielles, et donc augmenter leur temps de présence dans la mémoire. ■

En OCaml, il n'est pas possible d'effacer une donnée non mutable. Il est donc souhaitable de représenter les données les plus sensibles (par exemple, les clefs cryptographiques) par des structures de données mutables simples (tableaux ou chaînes de caractères), pour pouvoir les effacer facilement après usage. Cependant, cette solution pose un problème d'intégrité : une donnée mutable est *a priori* vulnérable à une attaque par modification.

Une solution complète passe donc par l'encapsulation : en regroupant dans un module la définition d'une structure mutable et toutes les opérations sur cette structure, et en n'exportant aucune fonction de modification, sauf une fonction d'effacement. Il est possible de stocker la donnée confidentielle dans une structure mutable (qui peut donc être effacée), tout en garantissant que le reste du programme la voit comme non-mutable (et ne peut donc pas la modifier).

> *Recommandation R-38: Encapsuler les manipulations de données confidentielles.*
> Pour chaque type de donnée confidentielle, il est recommandé d'encapsuler dans un module les fonctions qui manipulent ce type, de fournir dans ce module une fonction d'effacement, et de n'exporter le type de données que de manière abstraite. ■

Le système de finalisation semble être une bonne solution pour l'effacement des données confidentielles après usage, mais il s'agit d'une illusion dangereuse : en effet, le GC ne donne aucune garantie sur le temps écoulé entre le moment où une donnée n'est plus utile et le moment où sa fonction de finalisation est appelée. Cette durée représente une fenêtre de vulnérabilité supplémentaire (et de durée arbitraire) par rapport à un effacement explicite de la donnée confidentielle

par le programme.

> *Recommandation R-39: Ne pas se reposer sur les fonctions de finalisation pour effacer les données confidentielles.*
> Le programme doit effacer les données confidentielles dès qu'il n'en a plus besoin, sans attendre que le GC la désalloue. ■

Chapitre 2

Modèle d'exécution natif

2.1 Compilation

Les phases de compilation spécifiques du **back-end natif** sont les suivantes :
- Optimisation (de lambda-code vers du code intermédiaire C--),
- Spécialisation (de C-- vers du code natif).

La phase d'optimisation est commune aux différentes architectures et est succinctement décrite dans ce chapitre. La différenciation entre les différentes architectures est faite lors de la spécialisation qui réalise la sélection des instructions, l'allocation des registres, la linéarisation du code, l'impression du code natif et l'appel de l'outil assembleur du système pour produire le code objet. La spécialisation ne pose pas de problème particulier de sécurité et n'est donc pas détaillée dans ce document. La figure 2.1 illustre les différentes sorties du *back-end* et les options correspondantes. L'allocation de pile est déléguée au système d'exploitation.

Le compilateur en code natif d'OCaml, `ocamlopt`, est un véritable compilateur natif n'incluant pas d'étape de production de bytecode.

2.1.1 Options de compilation

Cette section décrit les options de compilation du compilateur natif `ocamlopt` qui peuvent avoir une incidence sur la sécurité. Les options communes avec les autres modèles sont décrites en section 1.2.7. La liste complète des options des différentes commandes de compilation OCaml et leurs recommandations d'usage associées sont récapitulées dans la table 5.2.

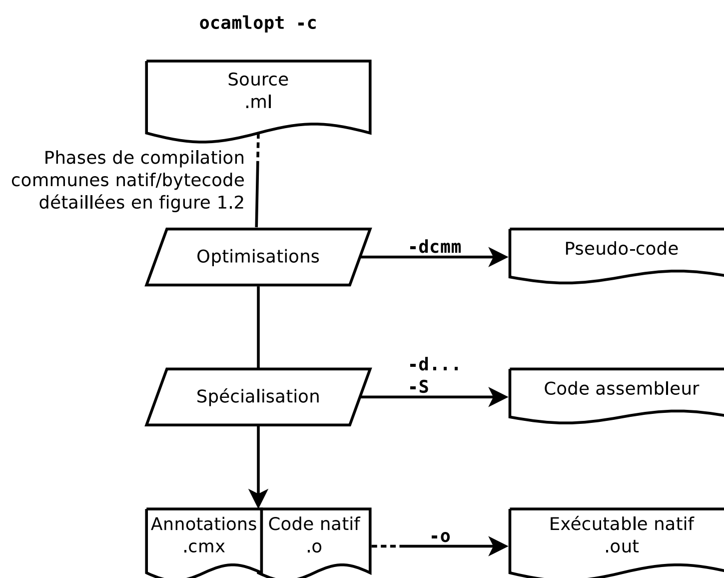


FIGURE 2.1 – Phases de compilation vers du code natif

`-g` Active l'enregistrement d'informations (correspondance entre programme compilé et source) dans le code compilé. Ces informations sont utilisées à l'exécution par le *backtrace*, voir section 2.3.2.

> *Recommandation R-40: Ne pas utiliser le mode* backtrace *en exploitation.*

`-p` Active le profilage d'exécutions. Le profilage peut servir à repérer les possibilités d'attaques par mesure du temps de calcul. Cette option n'est à utiliser qu'en phase de développement.

2.1.2 Optimisation

Le *back-end* en mode natif réalise des passes d'optimisation qui, entre autres, peuvent enlever du code mort, faire de l'*inlining*, réduire la portée d'identificateurs locaux ou propager des constantes.

La réduction de la portée d'une variable en accord avec ses différentes utilisations est une des optimisations effectuées par le compilateur natif. Ceci a pour effet de réduire la portée d'une expression définissant par exemple une variable locale `x` si la variable n'est utilisée que dans une partie des expressions couvertes

par la déclaration locale.

Dans l'exemple suivant : `let x = e1 in e2; e3; e4`, si la variable `x` n'est utilisée que dans l'expression `e2`, alors elle est libérée avant les expressions `e3` et `e4`, et sa valeur potentiellement désallouée par le GC. Ceci a pour effet de libérer la valeur plus tôt et donc d'éviter la rétention de données.

> *Intérêt I-18: Les optimisations n'invalident pas les garanties du typage.*

2.1.3 Compilation séparée, fichiers objet `.cmx`, `.cmxs` et `.cmxa`

La compilation des interfaces (extension `.mli`) est décrite en section 1.2.6. La compilation de chaque fichier d'implémentation `.ml` produit un **fichier objet natif** `.o` (`.obj` sous Windows) contenant le code compilé en natif, un fichier objet d'annotations `.cmx` et un fichier objet d'interface `.cmi` s'il n'existe pas de fichier d'interface `.mli`.

Un **fichier objet d'annotations** `.cmx` contient les informations utiles pour la liaison et l'optimisation du code natif produit (fichier `.o`/`.obj`). Il contient un certain nombre d'informations dont :

- Le nom de l'unité de compilation
- La liste des unités implémentées,
- La liste des unités requises (hachage MD5 des interfaces et des implémentations),
- La description sommaire de la structure de l'implémentation (pour chaque fonction, nom, arité et point d'entrée dans le fichier `.o`/`.obj`) utilisée par le compilateur pour la propagation de constantes et l'*inlining* entre modules.

L'option de compilation `-shared` permet de créer un **fichier objet natif pour le chargement dynamique** d'extension `.cmxs`. Les fichiers `.cmxs` sont des bibliothèques dynamiques de C (`.so`) avec des informations supplémentaires (nom du module et hachages MD5 des interfaces importées) pour l'éditeur de liens dynamiques de OCaml.

La table des symboles du fichier objet natif `.o`/`.obj` compilé par OCaml contient les symboles de chaque fonction du programme (fonctions nommées et anonymes), qu'elles soient cachées par encapsulation ou non.

L'option de compilation `-a` permet de créer une bibliothèque d'unités de compilation composée d'un **fichier objet d'annotations de bibliothèque** `.cmxa`

et un fichier natif de bibliothèque `.a`/`.lib`. En plus des informations sur chaque élément de la bibliothèque, le fichier `.cmxa` contient des informations relatives à l'édition de liens : liste des fichiers objet C requis et options particulières pour la l'éditeur de liens. L'option `-noautolink` permet d'ignorer ces informations.

2.2 Édition de liens et chargement

L'édition de liens et le chargement se font à la compilation ou en cours d'exécution (en cas de chargement dynamique) selon les options de compilation. Les deux possibilités sont détaillées ci-dessous et sont illustrées par la figure 2.2.

2.2.1 Chargement statique

Mis à part les chargements dynamiques explicites (appels au module `Dynlink`), l'édition de liens et le chargement se font toujours à la compilation ou au démarrage (cf. section 1.3.2) pour le runtime OCaml et les bibliothèques standard, externes et utilisateur, écrite en OCaml ou C.

2.2.2 Chargement dynamique de code natif OCaml

Le module `Dynlink` permet de charger dynamiquement du code natif compilé, cf. section 1.3.4. Un code doit être compilé avec l'option `-shared` pour être chargé dynamiquement. La seule vérification effectuée au chargement dynamique est celle de la cohérence des hachages MD5 des interfaces (cf. section 2.1.3) et celle des fonctions de configuration `allow_only` et `prohibit` de `Dynlink` (voir section 1.3.4). Le code chargé est un code natif n'incluant aucune information de typage. Il n'est pas possible de vérifier dynamiquement ce code.

> *Danger D-29: Absence de garanties du chargement dynamique de code natif.*

Les fichiers objet et les bibliothèques compilées sans l'option `-shared` ne peuvent être liés que statiquement, et ne peuvent pas être chargés dynamiquement par un programme OCaml.

Le chargement dynamique de code natif OCaml est effectué par appel au système d'exploitation. C'est donc celui-ci qui se charge d'éventuelles protections mémoire (en écriture et en exécution) de manière transparente pour OCaml.

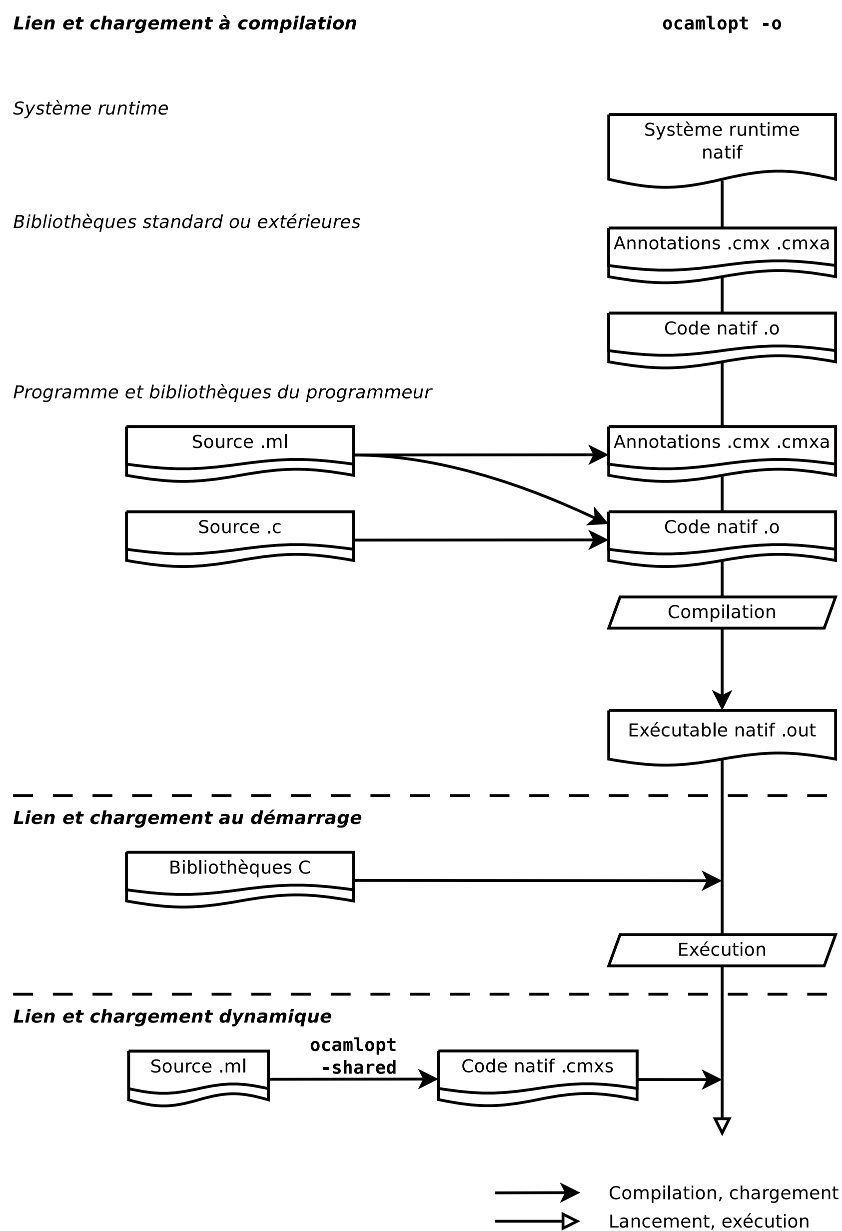


FIGURE 2.2 – Lien et chargement de code du modèle natif

Comme décrit dans la section 1.3.4, le contrôle des appels externes proposé par la fonction `allow_unsafe_modules` de `Dynlink` n'a aucun effet en natif (cf. danger D-19).

2.2.3 Interfaçage avec du code C

Comme expliqué dans la partie décrivant la construction `external` en section 1.2.5, du code natif externe peut être utilisé par un code OCaml. Il est chargé statiquement comme illustré par la figure 2.2. La compilation de l'exécutable effectue l'édition de liens entre les codes OCaml et C et charge le code natif C.

> *Intérêt I-19: La compilation en mode natif peut créer un exécutable indépendant.*
> Pour créer un exécutable indépendant il suffit de compiler le programme avec l'option `-cclib -static`. ∎

Le comportement par inclusion (option `-output-obj`) produit un fichier objet natif configuré de telle manière qu'il soit accessible par du code C. Cela permet de créer des bibliothèques pour C.

2.3 Exécution et interaction avec le système

2.3.1 Traitement des signaux

Cette section complète la section commune 1.4.1 et indique les particularités de la vérification de présence des signaux en natif. En cas d'émission de signal, le gestionnaire des signaux positionne le pointeur d'allocation de manière à faire croire qu'il n'y a plus d'espace dans le tas. À la prochaine allocation, la réponse au signal est déclenchée. La vérification se fait donc à l'allocation, ceci implique que si le programme boucle sans allouer, aucun signal émis ne sera traité.

> *Danger D-30: Pas de gestion de signal dans une boucle n'allouant pas.*

De la même manière, si une primitive C travaillant sur le tas boucle (comme un test d'égalité sur une valeur récursive), aucun signal émis ne sera traité.

> *Danger D-31: Pas de gestion de signal dans une primitive C.*

2.3.2 *Backtrace*, pile de levée d'exception

Le modèle natif ne possède pas de mode débogue à proprement parler mais l'option -g active l'enregistrement de la pile de levée d'exception ou *backtrace* qui est détaillé dans les paragraphes de la section 3.3.3 consacrés au *backtrace* du mode bytecode.

> *Recommandation R-41: Ne pas utiliser le mode* backtrace *en exploitation.*

Chapitre 3

Modèle d'exécution par machine virtuelle

Les intérêts principaux du modèle d'exécution par machine virtuelle ou modèle bytecode sont la rapidité de sa compilation (puisqu'aucune optimisation n'est réalisée) et son mode débogue. Un même bytecode OCaml peut être exécuté par toute machine virtuelle OCaml quelle que soit la plateforme. Comme indiqué par l'intérêt I-1, le comportement d'un programme exécuté sur machine virtuelle ne se distingue pas d'une exécution du même programme compilé en natif.

3.1 Compilation

Le **back-end du modèle bytecode** ne fait que produire le bytecode (illustré par la figure 3.1).

3.1.1 Options de compilation

Cette section décrit les options de compilation du compilateur bytecode **`ocamlc`** qui peuvent avoir une incidence sur la sécurité. Les options communes avec les autres modèles sont décrites en section 1.2.7. La liste complète des options des différentes commandes de compilation OCaml et leurs recommandations d'usage associées sont récapitulées dans la table 5.2.

`-custom` Inclut une copie de l'exécutif dans le fichier exécutable produit par le compilateur bytecode. Cette inclusion fait que l'exécution de l'exécutable produit ne dépend pas d'un fichier externe pour sa machine virtuelle

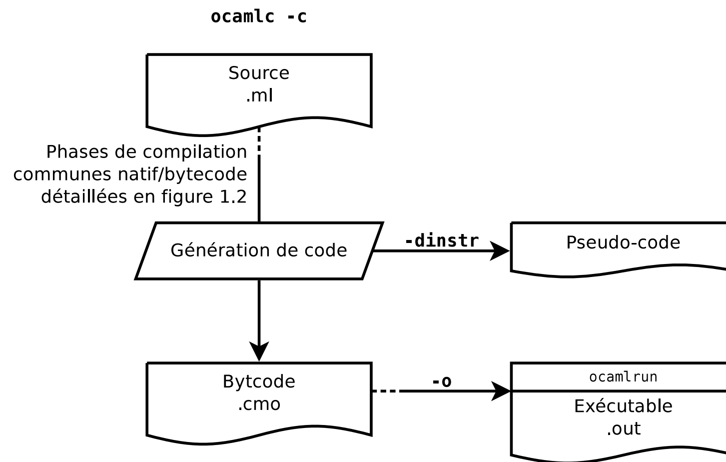


FIGURE 3.1 – Phases de compilation vers du bytecode

ocamlrun ni pour ses bibliothèques externes C. L'exécutable est spécialisé pour une architecture.

> *Intérêt I-20: L'option* **-custom** *crée un exécutable indépendant.*

> *Danger D-32: Un exécutable bytecode ne bénéficie pas des mêmes protections qu'un exécutable natif.*
> Contrairement à un exécutable natif dont le code est intégralement chargé en zone de code (cf. intérêt I-14), seul l'exécutif est protégé par le système, le bytecode étant placé dans le tas C. ■

-dllib <lib> et -dllpath <dir> Indique quelles bibliothèques dynamiques seront chargées dynamiquement et dans quels répertoires les chercher.

> *Danger D-33: Un chemin de recherche mal choisi peut conduire à une vulnérabilité.*

> *Recommandation R-42: Contrôler les répertoires de recherche pour éviter le chargement de bibliothèques malveillantes.*

-g Active le mode débogue qui rajoute des informations (correspondance entre programme compilé et source, informations de types) au code compilé.

Ces informations sont utilisées par le débogueur et le *backtrace* (cf. section 3.3.3).

> *Recommandation R-43: Ne pas utiliser le mode débogue en exploitation.*
> Il est déconseillé d'utiliser cette option sur le code en exploitation car elle facilite l'observation fine du programme (accès au variables manipulées par le programme). ■

`-output-obj` Génère un fichier C ou un fichier objet `.o` ou une bibliothèque dynamique `.so` contenant le bytecode dans un tableau d'entiers constant.

3.1.2 Bytecode OCaml

Le compilateur bytecode n'effectue pas d'optimisations pour ne pas augmenter le temps de compilation. L'absence d'optimisations ne change pas le comportement du programme, qui demeure identique à celui qui serait obtenu par compilation native.

Machine abstraite ZAM

Le langage bytecode OCaml a pour vocation d'être exécuté sur la machine abstraite de OCaml appelée ZINC Abstract Machine (ZAM) [Leroy, 1990]. Un bytecode est composé d'une suite d'instructions du jeu d'instructions de la ZAM. Le bytecode n'est soumis à aucun mécanisme de vérification à l'exécution.

La ZAM est composée d'une pile et de registres dont un accumulateur, un environnement et un pointeur de code. La ZAM exécute un programme qui est une suite d'instructions bytecode. Le GC est chargé de la gestion (cf. section 1.5.2) des valeurs placées dans le tas par la ZAM.

Langage du bytecode

Le langage du bytecode OCaml est composé de 146 instructions dont certaines sont des versions particularisées d'une instruction ou des compositions d'autres instructions pour obtenir une meilleure efficacité du bytecode. L'ensemble des fonctionnalités de la machine est décrit par un jeu de 64 instructions, qui offrent :

- Manipulations de la pile et des registres (i.e. `ACC`, `PUSH`),
- Opérations arithmétiques (i.e. `ADDINT`, `NEGINT`),

- Branchements (i.e. BRANCH, SWITCH qui met en œuvre le filtrage),
- Allocations de blocs (i.e. CLOSURE, MAKEBLOCK),
- Appels et retours (i.e. APPLY, RETURN, APPTERM pour les appels terminaux),
- Manipulation de l'environnement (i.e. ENVACC),
- Accès aux méthodes d'objet (i.e. GETMETHOD, GETDYNMET),
- Opérations de débogue (EVENT, BREAK),
- Vérification des signaux (CHECK_SIGNALS),
- Appels de primitives C (C_CALLN),
- Gestion des exceptions (i.e. PUSHTRAP, RAISE).

Le document [Clerc, 2010] propose une description complète du jeu d'instructions de la ZAM.

Appels à des fonctions C

Le fichier objet bytecode contient des appels à des fonctions C précompilées de l'exécutif (instructions C_CALLN).

Si le programme source ne contient pas de constructions non-sûres, toute construction non-sûre figurant dans le bytecode provient d'un appel fait par la bibliothèque standard.

Certaines constructions non-sûres (invalidant l'analyse statique, cf. section 1.2.5) apparaissant dans le texte source se traduisent par des appels de fonctions spécifiques dans le bytecode mais ce n'est pas une généralité :

- Les fonctions `Array.unsafe_*` se retrouvent dans le bytecode en instructions d'appel de code natif `caml_array_unsafe_*`.
- Les fonctions `String.unsafe_*` se retrouvent mais sous la forme d'instructions `*stringchar` ne comportant pas le mot `unsafe`.
- Les appels à la fonction `Marshal.to_string` se traduisent par l'instruction `caml_output_value_to_string`.
- Les appels aux fonctions de désérialisation de `Marshal` ne se traduisent pas par des instructions particulières.
- Les appels à la fonction `Obj.magic` ne donnent lieu à aucune instruction dans le code compilé.

Danger D-34: La relecture de bytecode est insuffisante pour garantir des propriétés de sécurité.

Recommandation R-44: L'analyse de programme doit être faite au niveau des textes source.

3.1.3 Compilation séparée, fichiers objet `.cmo` et `.cma`

La compilation des interfaces (extension `.mli`) est décrite en section 1.2.6. La compilation de chaque fichier d'implémentation `.ml` produit un fichier objet d'implémentation `.cmo` et un fichier objet d'interface s'il n'existe pas de fichier d'interface `.mli`.

Un **fichier objet d'implémentation** `.cmo` est une unité de compilation bytecode. Il est composé d'un numéro identifiant la version du format `.cmo`, du bytecode de l'unité de compilation, d'informations de débogue s'il est compilé avec l'option `-g` et d'une description de l'unité de compilation. Cette description fournit le nom de l'unité de compilation, la taille des blocs, la liste des relocations (références et définitions globales, primitives C et constantes structurées) pour l'édition de liens, la liste des importations requises (liste de paires composées du nom de l'unité et de l'empreinte MD5 de l'interface), la liste des primitives nécessaires (définies par la construction `external`), un marqueur indiquant si la liaison du code est requise même s'il n'est pas référencé (marqueur activé par l'option `-linkall`), ainsi que la position et la taille des informations de débogue.

L'option de compilation `-a` permet de créer un **fichier objet de bibliothèque** `.cma` qui est une collection d'unités de compilation. En plus du bytecode et des informations de chaque élément de la bibliothèque, le fichier `.cma` contient des informations relatives à la compilation particularisée (option `-custom`) : liste des fichiers objet C requis, options particulières pour la compilation C, bibliothèques dynamiques C requises.

3.2 Édition de liens et chargement

3.2.1 Chargement statique

Le compilateur `ocamlc` effectue la compilation puis l'édition de liens et le chargement de fichiers de bytecode `.cmo` ou `.cma`. Le chargement des primitives externes C est fait au démarrage ou à la compilation. Ces différences sont illustrées par la figure 3.2 et sont détaillées ici :

– Le comportement par défaut (sans l'option `-custom`) n'effectue le chargement dans le code que des bibliothèques OCaml. `ocamlrun` est appelé au lancement, il effectue le chargement des bibliothèques C requises. Ce

comportement par défaut est portable car le fichier bytecode exécutable produit ne dépend pas de la plateforme.
- Le comportement *particularisé* (option `-custom`) lie et charge dans le code l'exécutif OCaml et toutes les bibliothèques (OCaml ou C) à la compilation.
- Le comportement par inclusion (option `-output-obj`) qui place le bytecode dans un tableau d'entiers constant dans un fichier C (source ou compilé) permet un chargement du programme depuis un programme C. La bibliothèque `libcamlrun.a` fournit les exécutifs OCaml pour interpréter ce bytecode.
- Seuls les chargements dynamiques explicites (appels au module `Dynlink`) sont chargés en cours d'exécution.

Intérêt I-21: `ocamlrun` *ne charge dynamiquement du bytecode que par utilisation de* `Dynlink`*.*

Il est donc recommandé d'éviter l'utilisation de `Dynlink` (cf. recommandation R-28 et section 1.3.4).

3.2.2 Chargement dynamique de bytecode OCaml

Le chargement dynamique de code s'effectue par appels au module `Dynlink`, cf. section 1.3.4. Contrairement au code natif, le bytecode ne requiert pas de compilation particulière pour être chargé dynamiquement. On peut donc charger dynamiquement n'importe quel fichier objet à condition que les fichiers `.cmo` ou `.cma` soient du bon format.

La seule vérification effectuée au chargement dynamique est celle de la cohérence des hachages MD5 des interfaces (cf. section 2.1.3) et celle des fonctions de configuration `allow_only`, `prohibit` et `allow_unsafe_modules` de `Dynlink` (voir section 1.3.4). Le code chargé est un bytecode n'incluant aucune information de typage. Il n'est pas possible de vérifier dynamiquement ce code.

Danger D-35: Absence de garanties du chargement dynamique de bytecode.

Il est recommandé d'éviter l'utilisation de `Dynlink` (cf. recommandation R-28).

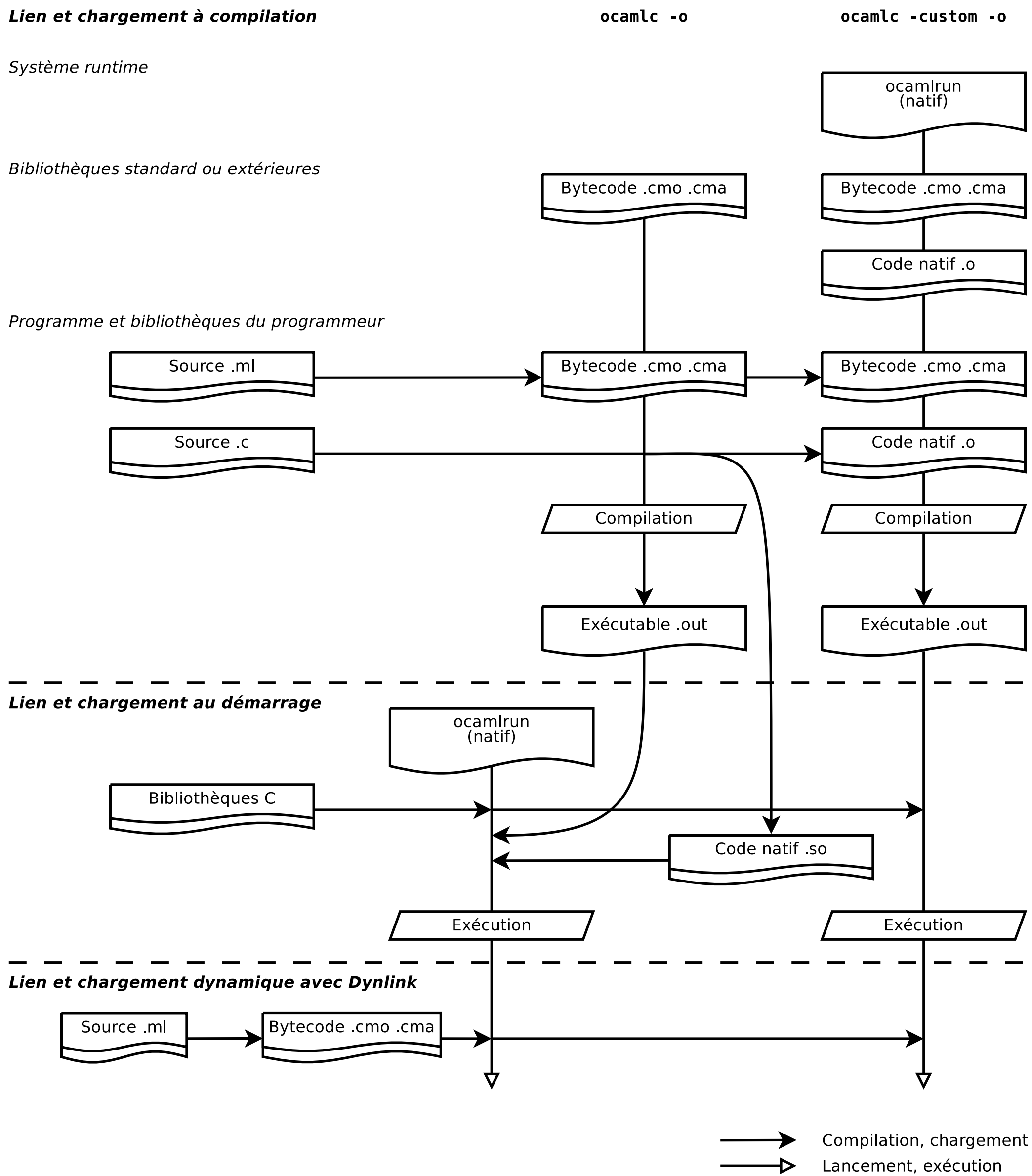


FIGURE 3.2 – Lien et chargement de codes du modèle par machine virtuelle

3.2.3 Interfaçage avec du code C

Comme expliqué dans la partie 1.2.5, du code natif C peut être utilisé par un code OCaml. Il est par défaut **chargé au démarrage** du programme par l'exécutif OCaml. L'option `-custom` du compilateur permet de créer une nouvelle machine virtuelle de bytecode avec le code C chargé statiquement. Cette différence apparaît dans la figure 3.2.

Le chargement de bibliothèques au démarrage du programme est contrôlé par l'option `-I` de `ocamlrun`, la variable d'environnement `CAML_LD_LIBRARY_PATH`, le fichier standard `$OCAMLLIB/ld.conf` ou `$CAMLLIB/ld.conf` ou le mécanisme par défaut du système d'exploitation (généralement `/etc/ld.so.conf` et `LD_LIBRARY_PATH` sous Unix, `PATH` sous Windows).

> *Danger D-36: Modification mal intentionnée du chemin de recherche des bibliothèques externes.*

> *Recommandation R-45: Contrôler les chemins de recherche des bibliothèques externes.*

3.3 Exécution et interaction avec le système

3.3.1 Machine virtuelle `ocamlrun`

La machine virtuelle OCaml n'effectue pas de compilation du bytecode comme c'est le cas pour les compilations JIT (Just-In-Time) de la Java Virtual Machine (JVM) et de la machine .NET. La compilation JIT répond à des besoins de performance que OCaml propose via son modèle d'exécution native (cf. section 2) avec compilation AOT (Ahead-Of-Time).

`ocamlrun` prend en argument le fichier objet exécutable à exécuter. Après son lancement, il n'est pas possible de charger de nouveaux fichiers objet sauf par l'utilisation de `Dynlink` dans le programme initialement chargé. Le chargement de code au démarrage par la machine virtuelle est décrit en section 3.2.

Les options de la machine virtuelle `ocamlrun` qui ont une incidence sur la sécurité sont les suivantes :

`-b` En mode débogue, permet d'accéder à une description de la pile d'exécution en cas de levée d'exception (voir la section 3.3.3),

`-I <dir>` Indique les répertoires dans lesquels les bibliothèques externes doivent être cherchées (voir la section 3.2.3 pour plus de détails).

3.3.2 Traitement des signaux

Cette section complète la section commune 1.4.1 et indique les particularités de la vérification de présence des signaux en bytecode.

À chaque instruction `APPLY`, `POPTRAP` et `CHECK_SIGNALS`, l'interpréteur de bytecode vérifie qu'aucun signal n'a été émis. Si un signal a été émis, le gestionnaire de signaux est appelé.

Si une primitive C travaillant sur le tas boucle (par exemple, un test d'égalité sur une valeur récursive), aucun signal émis n'est traité.

> *Danger D-37: Pas de gestion de signal dans une boucle d'une primitive C.*

3.3.3 Mode débogue

Le mode débogue comporte deux volets :
- un mode de compilation particulier, obtenu par l'option `-g`, produisant du bytecode dans lequel chaque point du programme est annoté (instruction `event`) avec des informations de localisation.
- un mode d'exécution avec `ocamlrun`, dirigé par le débogueur `ocamldebug`, qui ne peut être appliqué que sur du bytecode.

Exécution en mode débogue

L'exécution en mode débogue permet d'accéder aux données manipulées et aux informations de débogue. On pourrait penser que seuls les programmes compilés en mode débogue peuvent être exécutés en mode débogue mais ce n'est pas le cas. En effet, la variable d'environnement `CAML_DEBUG_SOCKET` permet de déterminer sur quel port le programme à déboguer doit communiquer avec `ocamldebug`. L'exécution d'un programme bytecode, compilé en mode débogue ou non, avec ou sans l'option `-custom`, dans un environnement où la variable `CAML_DEBUG_SOCKET` est définie, a pour effet de lancer l'exécution du programme en mode débogue : le programme attend des instructions de débogue sur le port correspondant.

Danger D-38: Utilisation de la variable `CAML_DEBUG_SOCKET` *pour observer une exécution bytecode.*
Il est possible d'exécuter un programme bytecode en mode débogue en définissant la variable `CAML_DEBUG_SOCKET`. Ceci permet d'observer les valeurs manipulées par le programme et donc de profiter d'une élévation de privilège pour accéder à des données confidentielles, ainsi que de rejouer tout ou partie du programme.■

Recommandation R-46: Préférer une exécution native pour empêcher une exécution en mode débogue du programme.
La présence de la variable d'environnement `CAML_DEBUG_SOCKET` est testée avant la première instruction du programme, celui-ci ne peut donc pas intervenir sur cette variable pour empêcher l'activation du mode débogue.

Pour empêcher une exécution en mode débogue du programme activé par la variable d'environnement `CAML_DEBUG_SOCKET`, il est conseillé de choisir le mode d'exécution natif dont l'exécutif n'inclut pas de mode débogue. ■

Backtrace, pile de levée d'exception

La **pile de levée d'exception** ou **backtrace** est la portion de la pile d'appels entre la levée et le rattrapage d'une exception. Elle peut être enregistrée si le programme a été compilé en mode débogue et si une des conditions suivantes est vérifiée :

- `ocamlrun` est lancé avec l'option `-b`,
- la fonction `record_backtrace` est appelée avec l'argument `true`, ou
- la variable d'environnement `OCAMLRUNPARAM` possède le paramètre `b`.

La pile de levée d'une exception est accessible au rattrapage de l'exception via les fonctions `get_backtrace`, `print_backtrace` du module `Printexc`.

Danger D-39: Fuite d'information par la pile de levée d'exceptions.
L'accès à la pile de levée d'une exception peut potentiellement révéler des informations sur les valeurs manipulées par le programme. ■

Il est important de ne pas utiliser le mode débogue de compilation pour créer le bytecode qui sera mis en exploitation (cf. recommandation R-43).

Chapitre 4

Modèle d'exécution par boucle interactive

La **boucle interactive `ocaml`** ou **toplevel** permet une exécution pas à pas d'un programme OCaml. Son fonctionnement est celui d'une boucle REPL (Read-Eval-Print Loop) : pour chaque phrase entrée, elle effectue les phases de *parsing*, de typage et d'évaluation puis affiche les valeurs, les types inférés et effectue les entrées/sorties standard. Elle est essentiellement utilisée pour l'enseignement et la mise au point de programmes. Le fonctionnement de la boucle interactive est illustré par la figure 4.1. Le *toplevel* `ocaml` permet aussi une évaluation en mode script pour lequel les phases *parsing*–typage–évaluation sont faites sur l'ensemble d'un texte source.

Les exécutions en mode interactif ou en mode script ne sont pas adaptées à une exécution en exploitation. L'exécution en mode script d'une application composée de plusieurs fichiers source d'implémentation ne bénéficie pas des garanties apportées par la compilation séparée (cf. section 1.2.6) et par la liaison des codes compilés (cf. section 1.3). En mode interactif, toutes les données manipulées sont très facilement accessibles. De plus, le fil d'exécution peut être stoppé après n'importe quelle évaluation intermédiaire et prolongé de n'importe quelle manière.

> *Recommandation R-47: Ne pas utiliser le* toplevel ***`ocaml`*** *en exploitation.*

L'outil `ocamlmktop` permet de créer une boucle interactive spécialisée préchargeant des modules spécifiés. De plus, l'option `-init <file>` évalue le contenu

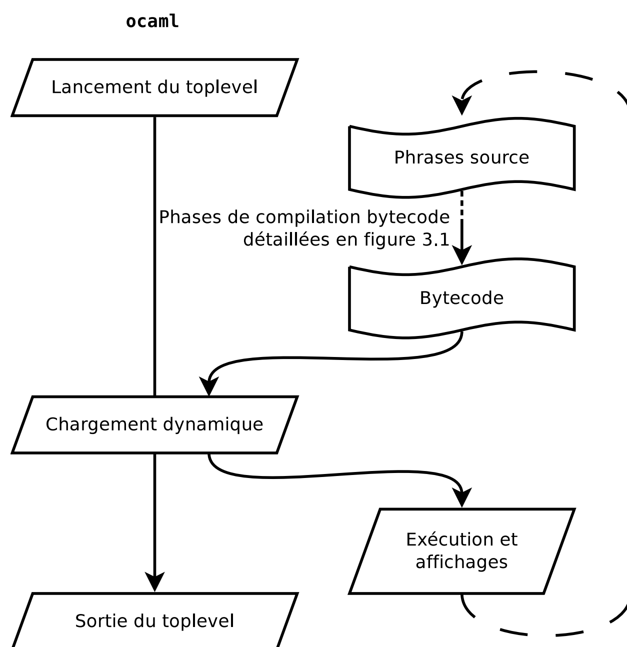


FIGURE 4.1 – Boucle interactive

du fichier `file` à l'initialisation de la boucle interactive. Si cette option n'est pas fournie et si un fichier nommé `.ocamlinit` est présent dans le répertoire courant, alors le contenu de ce fichier est évalué à l'initialisation de la boucle. Il faut donc contrôler la présence de fichiers `.ocamlinit`. Les options communes au modèle bytecode sont décrites en section 1.2.7 ou section 3.1.1. La liste complète des options des différentes commandes de compilation OCaml et leurs recommandations d'usage associées sont récapitulées dans la table 5.2.

Chapitre 5

Interfaçage avec l'environnement système

5.1 Variables d'environnement

Des variables d'environnement sont utilisées par OCaml pour certaines configurations. En voici la liste ci-dessous.

– Les variables `CAML_LD_LIBRARY_PATH`, `OCAMLLIB` et `CAMLLIB` (fichiers `$OCAMLLIB/ld.conf` et `$CAMLLIB/ld.conf`) déterminent le chemin de recherche des bibliothèques chargées au démarrage du programme (cf. 3.2.3).
– La variable `PATH` indique d'une part au *shell* où trouver le fichier de bytecode exécutable correspondant à une commande donnée et d'autre part à **ocamlrun** où trouver ce même fichier pour charger le bytecode. Dans le cas d'un bytecode particularisé, la variable est utilisée pour retrouver le fichier exécutable et charger le bytecode.
– La variable `OCAMLRUNPARAM` permet de configurer le GC (cf. section 1.5.2), de la machine virtuelle **ocamlrun** (cf. section 3.3.1) et de l'exécutif natif. Excepté le paramètre `b` vu dans la partie consacrée au *backtrace* de la section 3.3.3 (concernant aussi le débogue des exceptions en natif, cf. section 2.3.2), les paramètres de la variable d'environnement `OCAMLRUNPARAM` ne permettent pas d'influencer le comportement du programme de manière significative pour la sécurité.
– La variable `CAML_DEBUG_SOCKET` détermine sur quel port le programme à déboguer doit communiquer avec le débogueur (cf. section 3.3.3).
– La variable `TMPDIR` détermine le répertoire dans lequel les fichiers tempo-

raires sont créés par les fonctions du module `Filename`[1]. Dans le cas d'un programme ayant des privilèges particuliers, il est recommandé de vérifier que cette variable a bien une valeur acceptable.
– La variable `CAMLSIGPIPE`, utilisée sous Windows, donne un canal de communication pour simuler les signaux Unix.
– La variable `TERM` indique le type de terminal sur lequel le programme est exécuté, elle est passée aux fonctions de la bibliothèque `termcap`. Elle ne pose pas de problème de sécurité spécifique à OCaml.
– La variable `CAML_DEBUG_FILE` permet d'afficher les piles de levée d'exceptions (cf. section 3.3.3) pour un programme bytecode ayant été compilé par inclusion (option `-output-obj` du compilateur bytecode, voir la section 3.1.1), même si le programme n'a pas été compilé en mode débogue. Le fichier indiqué par la variable contient les informations de débogue. Ce fichier est compilé avec les options `-g -output-obj`.

5.2 Privilèges système

Il est à rappeler tout d'abord que les fichiers exécutables que sont le compilateur et les préprocesseurs n'ont pas besoin de disposer de droits privilégiés.

> *Recommandation R-48: Ne pas donner de droits privilégiés aux compilateurs et préprocesseurs.*

Les différents modèles d'exécution ont des implications différentes vis-à-vis des privilèges système.

En modèle d'exécution natif, les droits d'exécution sont ceux de l'exécutable.

En modèle d'exécution bytecode, les implications sont différentes selon les options de compilation (comme illustré par la figure 3.2 de la section 2.2).
– En compilation bytecode par défaut (sans l'option `-custom`), l'appel à **`ocamlrun`** est inclus dans le fichier exécutable produit (entête **`#!/../ocamlrun`**). Les droits d'exécution du programme dépendent du système d'exploitation : ce sont ceux de **`ocamlrun`** ou ceux du fichier produit.

> *Recommandation R-49: Ne pas donner de droits privilégiés à* ***`ocamlrun`****.*
>
> La machine virtuelle **`ocamlrun`** pouvant exécuter n'importe

1. Le compilateur OCaml lui-même utilise ces fonctions pour ses fichiers temporaires.

quel bytecode OCaml, il est conseillé de ne pas lui donner de droits privilégiés et donc de préférer les compilations natives ou bytecode *particularisées*. ■

– En compilation *particularisée* (avec option **-custom**), l'exécutable produit inclut une copie de la machine virtuelle, il possède donc ses propres droits d'exécution.

Danger D-40: Détournement d'un bytecode particularisé.
Un bytecode particularisé contient une copie de la machine virtuelle qu'un attaquant peut forcer à exécuter un fichier de bytecode arbitraire. ■

Recommandation R-50: Ne pas donner de droits privilégiés à un exécutable bytecode particularisé.

Avec le modèle d'exécution par boucle interactive, les droits d'exécution sont ceux de la boucle **ocaml** ou dépendent du système d'exploitation en cas d'exécution par script. La situation est similaire si une boucle interactive spécialisée est créée avec **ocamlmktop**.

Recommandation R-51: Ne pas donner de droits privilégiés aux boucles interactives.

5.3 Différences d'exécution dûes aux environnements

Les modèles d'exécution OCaml sont largement compatibles : ils ont le même comportement et produisent les mêmes résultats. Cependant, ils diffèrent sur les points suivants relatifs à l'environnement d'exécution.

1. Les privilèges système d'un programme bytecode dépendent du système d'exploitation (cf. section 5.2).
2. La détection des signaux est faite à des moments différents dans le modèle natif (cf. section 2.3.1) et dans le modèle bytecode (cf. section 3.3.2).
3. Le rattrapage des débordements de pile (*stack overflow*) n'est pas toujours possible en natif : sur certains systèmes d'exploitation, la détection du

débordement de pile se traduit par un signal bas niveau d'erreur d'accès à la mémoire et non par la levée de l'exception `Stack_overflow`.

4. Le résultat des opérations sur les nombres flottants peut être différent suivant les architectures :
   – sur IA32, en modèle natif, certains calculs sont faits avec une précision supérieure à celle spécifiée par la norme IEEE754, ce qui peut provoquer des erreurs d'arrondis,
   – sur Alpha, dans les deux modèles, les flottants infinis ou dénormalisés peuvent arrêter le programme au lieu de continuer à calculer comme sur les autres architectures.
5. Les *Threads* ne sont pas toujours disponibles suivant le système, elles peuvent être émulées en modèle bytecode (cf. section 1.4.2).

5.4 Récapitulatifs

5.4.1 Extensions de fichiers

La table 5.1 récapitule les différents extensions de fichiers pour OCaml.

	Fichiers source
`.mli`	Texte source d'interface OCaml
`.ml`	Texte source d'implémentation OCaml
	Fichiers objet
`.cmi`	Interface OCaml compilée
`.cmx`	Annotations OCaml de code natif
`.o`	Code natif
`.cmxa`	Bibliothèque native OCaml
`.cmxs`	Code natif OCaml pour chargement dynamique
`.cmo`	Bytecode OCaml
`.cma`	Bibliothèque bytecode OCaml

Table 5.1 – Extensions de fichiers pour OCaml

5.4.2 Options de compilation et de la boucle interactive

La table 5.2 récapitule les recommandations concernant les options des trois compilateurs **ocamlopt**, **ocamlc** et **ocaml**. Les deux phases de vie du logiciel auxquelles elles peuvent être associées sont aussi signalées : mise au point et exploitation.

Les discussions portant sur chacune des options considérées sont dans les sections 2.1.1, 3.1.1 et le chapitre 4.

Commande	**Option**	**Phase**	
ocamlopt ocamlc ocaml		Mise au point	Exploitation
●●○	-annot	✚	
○●○	-custom	✚	✚
○●○	-dllib <lib>	✶	✶
○●○	-dllpath <dir>	✶	✶
●●○	-g		▬
●●●	-I <dir>	✶	✶
○○●	-init <file>	✚	✚
●●○	-linkall	✶	✶
●●●	-noassert	▬	▬
●●○	-noautolink	✚	✚
●○○	-p		▬
●●○	-pp <command>	✶	✶
●●●	-rectypes	▬	▬
●●●	-strict-sequence	✚	✚
●●●	-unsafe	▬	▬
●●●	-w +a-4	✚	✚
●●●	-warn-error +a	✚	✚

TABLE 5.2 – Avis sur l'usage des options de compilation

Les options des commandes **ocamlopt**, **ocamlc**, **ocaml** dont l'usage peut avoir un impact sur la sécurité de l'application sont énumérées. Pour chacune les recommandations positives (✚), recommandations négatives (▬), et avertissements (✶) d'usage sont rappelés.

Bibliographie

[ANA-SECU, 2011] Analyse des langages OCaml, Scala et F#. Étude de la sécurité intrinsèque des langages fonctionnels (LaFoSec) L2.2.2, ANSSI (2011). Étude menée par un consortium composé de SafeRiver, CEDRIC, Normation et Oppida.

[Clerc, 2010] CLERC, X. (2010). Caml virtual machine : Instruction set. Disponible en ligne http://cadmium.x9c.fr/distrib/caml-instructions.pdf (dernière visite le 3 août 2011). Version 1.4 du document.

[Cuoq et Doligez, 2008] CUOQ, P. et DOLIGEZ, D. (2008). Hashconsing in an incrementally garbage-collected system : A story of weak pointers and hashconsing in OCaml 3.10.2. *In 2008 ACM SIGPLAN workshop on ML*, pages 13–22, New York. ACM.

[Doligez, 1995] DOLIGEZ, D. (1995). *Conception, réalisation et certification d'un glaneur de cellules concurrent*. Thèse de doctorat, Université Paris 7.

[Doligez et Gonthier, 1994] DOLIGEZ, D. et GONTHIER, G. (1994). Portable, unobtrusive garbage collection for multiprocessor systems. *In POPL*, pages 70–83.

[Doligez et Leroy, 1993] DOLIGEZ, D. et LEROY, X. (1993). A concurrent, generational garbage collector for a multithreaded implementation of ML. *In POPL*, pages 113–123.

[ETAT-LANG, 2011] État des lieux des langages fonctionnels. Étude de la sécurité intrinsèque des langages fonctionnels (LaFoSec) L2.1.2, ANSSI (2011). Étude menée par un consortium composé de SafeRiver, CEDRIC, Normation et Oppida.

[Garrigue, 2004] GARRIGUE, J. (2004). Relaxing the value restriction. *In* KAMEYAMA, Y. et STUCKEY, P. J., éditeurs : *FLOPS*, volume 2998 de *Lecture Notes in Computer Science*, pages 196–213. Springer.

[Garrigue et Rémy, 1999] GARRIGUE, J. et RÉMY, D. (1999). Semi-explicit first-class polymorphism for ML. *Information and Computation*, 155(1–2):134–169.

[Harrop, 2010] HARROP, J. (2010). Towards a mark-region gc for hlvm. Site Web du Flying Frog Blog. `http://flyingfrogblog.blogspot.com/2010/12/towards-mark-region-gc-for-hlvm.html` (dernière visite le 11 février 2010).

[Leroy, 1990] LEROY, X. (1990). The ZINC experiment : an economical implementation of the ML language. Technical report 117, INRIA.

[Leroy, 1994] LEROY, X. (1994). Manifest types, modules, and separate compilation. *In POPL*, pages 109–122.

[Leroy, 1995] LEROY, X. (1995). Applicative functors and fully transparent higher-order modules. *In POPL*, pages 142–153.

[Leroy, 1997] LEROY, X. (1997). The effectiveness of type-based unboxing. *In TIC*.

[Leroy, 1999] LEROY, X. (1999). Objects and classes versus modules in Objective Caml. Disponible à l'adresse `http://gallium.inria.fr/~xleroy/talks/icfp99.ps.gz` (dernière visite le 31 janvier 2011). Support de présentation invitée à ICFP.

[Leroy, 2000] LEROY, X. (2000). A modular module system. *Journal of Functional Programming*, 10(3):269–303.

[Leroy, 2005] LEROY, X. (2005). From Krivine's machine to the Caml implementations. Disponible à l'adresse `http://gallium.inria.fr/~xleroy/talks/zam-kazam05.pdf` (dernière visite le 31 janvier 2011). Support de présentation invitée à KAZAM workshop.

[Leroy *et al.*, 2010] LEROY, X., DOLIGEZ, D., FRISCH, A., GARRIGUE, J., RÉMY, D. et VOUILLON, J. (2010). The Objective Caml system release 3.12 – documentation and user's manual. INRIA. Disponible en ligne `http://caml.inria.fr/pub/docs/manual-ocaml/`.

[OUTILS-OCAML, 2011] Outils associés au langage OCaml. Étude de la sécurité intrinsèque des langages fonctionnels (LaFoSec) L3.1.2, ANSSI (2011). Étude menée par un consortium composé de SafeRiver, CEDRIC, Normation et Oppida.

[Pottier et Rémy, 2005] POTTIER, F. et RÉMY, D. (2005). The essence of ML type inference. *In* PIERCE, B. C., éditeur : *Advanced topics in types and programming languages*, chapitre 10, pages 389–489. MIT Press.

[Rémy et Vouillon, 1998] Rémy, D. et Vouillon, J. (1998). Objective ML : An effective object-oriented extension to ML. *TAPOS*, 4(1):27–50.

Table des figures

1.1 Principe général de la compilation 10
1.2 Phases communes de compilation 13
1.3 Exemple d'allocation 54
1.4 Exemple de partage 57
1.5 Structure de la mémoire 58

2.1 Phases de compilation vers du code natif 65
2.2 Lien et chargement de code du modèle natif 68

3.1 Phases de compilation vers du bytecode 72
3.2 Lien et chargement de codes du modèle par machine virtuelle . . 77

4.1 Boucle interactive 82

Table des tables

1.1 Liste des avertissements de compilation (option `-warn-help`) . . 42

5.1 Extensions de fichiers pour OCaml 86
5.2 Avis sur l'usage des options de compilation 87

Tables des avis de sécurité

Table des intérêts

I-1 Compatibilité des modèles d'exécution . . . 11
I-2 Mise à disposition des codes intermédiaires produits par le compilateur 13
I-3 L'outil `camlp4` peut servir à analyser le texte source . . . 16
I-4 Le typage facilite le développement d'applications . . . 16
I-5 Le typage garantit de nombreuses propriétés du texte source . . . 17
I-6 Le processus de compilation garantit la préservation des propriétés du texte source . . . 17
I-7 Le compilateur insère des vérifications dynamiques de propriétés dans le code compilé . . . 18
I-8 Compilation séparée assurée par le hachage MD5 des interfaces . . . 38
I-9 La vérification des hachages MD5 des interfaces garantit la cohérence du typage . . . 46
I-10 Garantie d'initialisation des pointeurs par non interférence entre traitement des signaux et GC . . . 50
I-11 Automatisation de la gestion mémoire . . . 51
I-12 Absence d'erreurs d'allocation mémoire . . . 52
I-13 Absence d'erreurs des accès en mémoire . . . 52
I-14 Séparation programme–données reflétée au bas niveau dans le modèle natif . . . 53
I-15 Absence d'erreurs de désallocation mémoire . . . 57
I-16 Libération de la mémoire par appel explicite au GC . . . 61
I-17 Non-duplication des arguments des fonctions . . . 61
I-18 Les optimisations n'invalident pas les garanties du typage . . . 66
I-19 La compilation en mode natif peut créer un exécutable indépendant . 69
I-20 L'option `-custom` crée un exécutable indépendant . . . 72

I-21 ocamlrun ne charge dynamiquement du bytecode que par utilisation de Dynlink . . . 76

Table des dangers

D-1 Introduction de source malveillant par l'utilisation de préprocesseurs . 14
D-2 Les informations de typage ne sont pas incluses dans le code compilé 18
D-3 Pas de détection de dépassement d'entier à l'exécution 19
D-4 L'exécution de la comparaison n'est pas typée 27
D-5 La comparaison de valeurs d'un type somme n'est pas complètement spécifiée . . . 27
D-6 Possibilité de contournement du typage et de l'encapsulation par la comparaison d'exceptions . . . 28
D-7 Le hachage ne connaît pas les limites de l'abstraction 28
D-8 L'utilisation du module Obj détruit la cohérence du texte source apportée par le typage . . . 29
D-9 La sérialisation/désérialisation peut donner accès à la zone mémoire utilisée par le programme . . . 30
D-10 Aucune vérification n'est faite sur les valeurs à désérialiser 31
D-11 Les accès à la mémoire effectués par du code externe ne sont pas contrôlés . . . 33
D-12 Invalidation des vérifications de débordement par les fonctions unsafe_* et l'option -unsafe . . . 34
D-13 Possibilité de contournement de l'analyse statique par utilisation de constructions non-sûres . . . 36
D-14 Substitution de bibliothèques par attaque des répertoires de recherche 39
D-15 Risque de manipulation malveillante de l'argument de Dynlink.loadfile . . . 45
D-16 Modification malveillante du fichier chargé 45
D-17 La vérification des hachages MD5 ne protège pas contre une modification délibérée du fichier chargé . . . 46
D-18 Risque de contournement du typage et de l'encapsulation par utilisation du module Dynlink . . . 46
D-19 Les appels externes d'un code natif chargé dynamiquement ne sont pas contrôlés par allow_unsafe_modules . . . 47
D-20 Re-configuration du gestionnaire de signaux par un texte source non-digne de confiance . . . 50

D-21 L'allocation des chaînes de caractères peut permettre une consultation illicite de la mémoire . . . 54
D-22 Le partage d'une valeur mutable peut aboutir à des modifications non voulues de cette valeur . . . 55
D-23 Mutabilité et partage de chaînes de caractères littérales . . . 56
D-24 Duplication en mémoire des données confidentielles . . . 58
D-25 Les données inutiles du tas mineur ne sont pas effacées . . . 58
D-26 Désallocation retardée . . . 59
D-27 Accès mémoire par code C non contrôlé . . . 60
D-28 Duplication de données par le GC . . . 62
D-29 Absence de garanties du chargement dynamique de code natif . . . 67
D-30 Pas de gestion de signal dans une boucle n'allouant pas . . . 69
D-31 Pas de gestion de signal dans une primitive C . . . 69
D-32 Un exécutable bytecode ne bénéficie pas des mêmes protections qu'un exécutable natif . . . 72
D-33 Un chemin de recherche mal choisi peut conduire à une vulnérabilité 72
D-34 La relecture de bytecode est insuffisante pour garantir des propriétés de sécurité . . . 74
D-35 Absence de garanties du chargement dynamique de bytecode . . . 76
D-36 Modification mal intentionnée du chemin de recherche des bibliothèques externes . . . 78
D-37 Pas de gestion de signal dans une boucle d'une primitive C . . . 79
D-38 Utilisation de la variable `CAML_DEBUG_SOCKET` pour observer une exécution bytecode . . . 80
D-39 Fuite d'information par la pile de levée d'exceptions . . . 80
D-40 Détournement d'un bytecode particularisé . . . 85

Table des recommandations

R-1 S'assurer de l'innocuité d'un préprocesseur avant son utilisation . . . 14
R-2 Préférer `camlp4` comme préprocesseur . . . 15
R-3 Utiliser `camlp4` pour adapter la syntaxe de chaque fichier `.ml` à son niveau de sécurité . . . 16
R-4 Ne créer un exécutable qu'à partir de textes source . . . 18
R-5 Ne pas oublier que l'arithmétique des entiers est une arithmétique modulaire . . . 19
R-6 Interdire le masquage des exceptions . . . 22

R-7 N'utiliser les fonctions de comparaison génériques que si la définition précise de l'ordre n'a pas d'importance . . . 28
R-8 Examiner l'utilisation des exceptions dans tout texte source non digne de confiance . . . 28
R-9 Ne pas utiliser le module `Obj` . . . 30
R-10 S'assurer de la non-utilisation du module `Obj` par les bibliothèques externes chargées . . . 30
R-11 Justifier et confiner toute utilisation du module `Obj` . . . 30
R-12 Définir des opérations robustes pour la transmission de données . . . 30
R-13 Éviter d'utiliser le module `Marshal` . . . 31
R-14 Éviter l'utilisation de la construction `external` . . . 33
R-15 Analyser les sources C avant leur utilisation . . . 33
R-16 Proscrire l'usage des fonctions `unsafe_*` et de l'option `-unsafe` . . . 34
R-17 Éviter l'emploi de constructions non-sûres . . . 36
R-18 Bien choisir les options de compilation pour préserver les garanties apportées par l'analyse statique . . . 37
R-19 Utiliser l'option `-annot` de manière à pouvoir vérifier la cohérence du texte source avec la spécification . . . 39
R-20 Contrôler les répertoires de recherche pour éviter le chargement de bibliothèques malveillantes . . . 39
R-21 Indiquer explicitement les bibliothèques à charger avec `-cclib` et interdire tout chargement automatique avec `-noautolink` . . . 40
R-22 Fournir explicitement les répertoires des bibliothèques utilisées, y compris celui de la bibliothèque standard . . . 40
R-23 Ne pas utiliser l'option `-rectypes` . . . 40
R-24 Utiliser la vérification stricte des séquences d'expressions . . . 40
R-25 En cas d'utilisation de `-linkall`, vérifier toutes les bibliothèques liées 43
R-26 Contrôler la provenance ou valider l'argument de `Dynlink.loadfile` 45
R-27 Contrôler les droits d'accès aux fichiers susceptibles d'être chargés dynamiquement . . . 45
R-28 Éviter les utilisations de `Dynlink` . . . 46
R-29 Ne pas se reposer entièrement sur le contrôle d'accès fourni par `Dynlink` 47
R-30 Proscrire les utilisations des fonctions `reset`, `add_interfaces` et `add_available_units` du module `Dynlink` . . . 47
R-31 Contrôler le chargement dynamique . . . 48
R-32 Contrôler les configurations du gestionnaire de signaux . . . 50
R-33 Exclure les modules `Sys` et `Unix` en cas de chargement dynamique . . 50
R-34 Interdire l'utilisation de `String.create` . . . 54

R-35 Forcer la copie des chaînes de caractères littérales 56
R-36 Vérifier que le partage ne porte que sur des valeurs n'ayant aucune sous-structure mutable 56
R-37 Limiter l'usage des bibliothèques C 60
R-38 Encapsuler les manipulations de données confidentielles 62
R-39 Ne pas se reposer sur les fonctions de finalisation pour effacer les données confidentielles 63
R-40 Ne pas utiliser le mode *backtrace* en exploitation 65
R-41 Ne pas utiliser le mode *backtrace* en exploitation 70
R-42 Contrôler les répertoires de recherche pour éviter le chargement de bibliothèques malveillantes 72
R-43 Ne pas utiliser le mode débogue en exploitation 73
R-44 L'analyse de programme doit être faite au niveau des textes source . 75
R-45 Contrôler les chemins de recherche des bibliothèques externes 78
R-46 Préférer une exécution native pour empêcher une exécution en mode débogue du programme 80
R-47 Ne pas utiliser le *toplevel* `ocaml` en exploitation 81
R-48 Ne pas donner de droits privilégiés aux compilateurs et préprocesseurs 84
R-49 Ne pas donner de droits privilégiés à **`ocamlrun`** 84
R-50 Ne pas donner de droits privilégiés à un exécutable bytecode particularisé 85
R-51 Ne pas donner de droits privilégiés aux boucles interactives 85

Acronymes

ANSI American National Standards Institute

ANSSI Agence Nationale de la Sécurité des Systèmes d'Information

AOT Compilation Ahead-Of-Time

AST Arbre de Syntaxe Abstraite

GC Garbage Collector (ramasse-miettes)

INRIA Institut National de Recherche en Informatique et en Automatique

JIT Compilation Just-In-Time

JVM Java Virtual Machine

POSIX Portable Operating System Interface pour UniX

REPL Read-Eval-Print Loop

SGDSN Secrétariat Général de la Défense et de la Sécurité Nationale

ZAM ZINC Abstract Machine (machine virtuelle de OCaml)

Index

back-end, 11
  bytecode, 71
  natif, 64
backtrace, 80
bloc, 20
bloc mémoire, 18
bootstrap, 11
boucle interactive, 81
bytecode, 10

camlp4, 15
camlp5, 15
chargement au démarrage par exécutif C, 43
chargement au démarrage par la machine virtuelle, 43, 78
chargement dynamique, 43
chargement statique, 43
code assembleur, voir code natif
code intermédiaire, 12
code machine, voir code natif
code natif, 9
code octet, voir bytecode
comparaisons de valeurs, 25
construction non-sûre, 29

dépassement d'entier, 19

exécutif, 48

fichier objet, 86
  .cma (bibliothèque, bytecode), 75
  .cmi (interface), 37
  .cmo (implémentation, bytecode), 75
  .cmxa (bibliothèque, natif), 66
  .cmxs (implémentation, natif dynamique), 66
  .cmx (annotations, natif), 66
  .o (natif), 66
fichier source, 86
  .mli (interface), 37
  .ml (implémentation), 9
front-end, 11, 12

garbage collector, voir GC
GC, 56
  finalisation, 60
  GC majeur, 59
  GC mineur, 58

interpréteur de bytecode, voir machine virtuelle

lambda-code, 12
lexing, 12

machine virtuelle, 10
macro, 15
mot de la machine, 18

ocaml, 10, 81

ocamlc, 10
ocamldebug, 79
ocamlmktop, 81
ocamlopt, 9
ocamlrun, 10
option de compilation, 87
  -I, 39, 78, 79
  -annot, 39
  -a, 66, 75
  -b, 78, 80
  -custom, 71, 76
  -dlambda, 13
  -dllib, 72
  -dllpath, 72
  -dparsetree, 13
  -drawlambda, 13
  -g, 65, 70, 72, 79
  -init, 81
  -linkall, 39, 43
  -noassert, 39
  -noautolink, 39
  -nostdlib, 40
  -output-obj, 69, 73, 76
  -pp, 14, 40
  -p, 65
  -rectypes, 40
  -shared, 66
  -strict-sequence, 40
  -thread, 51
  -unsafe, 29, 34, 37, 40
  -vmthread, 51
  -warn-error, 37, 41
  -warn-help, 42
  -w, 37, 41
overflow, voir dépassement d'entier

parsing, 12
pile, 52
pile de levée d'exception, voir backtrace
point d'entrée, 37
pointeur, 20
préprocesseur, 14

ramasse-miettes, voir GC

système runtime, voir exécutif

tas, 52
tas C, 57
tas Caml, 57
tas majeur, 57
tas mineur, 57
toplevel, voir boucle interactive
typage, 12

unité de compilation, 9

zone de code, 52